FORMAL SEMANTICS AND THE LOGICAL STRUCTURE OF PROGRAMMING LANGUAGES

A THESIS

Presented to

The Faculty of the Division of Graduate

Studies and Research

by

Richard A. DeMillo

In Partial Fulfillment

of the Requirements for the Degree

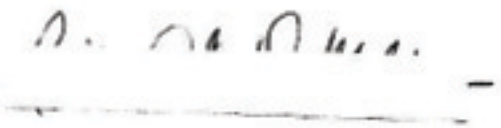
Doctor of Philosophy

in the School of Information and Computer Science

Georgia Institute of Technology

July, 1972

In presenting the dissertation as a partial fulfillment of the requirements for an advanced degree from the Georgia Institute of Technology, I agree that the Library of the Institute shall make it available for inspection and circulation in accordance with its regulations governing materials of this type. I agree that permission to copy from, or to publish from, this dissertation may be granted by the professor under whose direction it was written, or, in his absence, by the Dean of the Graduate Division when such copying or publication is solely for scholarly purposes and does not involve potential financial gain. It is understood that any copying from, or publication of, this dissertation which involves potential financial gain will not be allowed without written permission.



7/25/68

FORMAL SEMANTICS AND THE LOGICAL STRUCTURE

OF PROGRAMMING LANGUAGES

Approved:

Chairman: Lucio Chiaraviglio

Member: William I. Grosky

Member: John M. Gwynn, Jr.

Member: Michael D. Kelly

Visiting Member: Bas C. van Fraassen
Department of Philosophy
University of Toronto

Date approved by Chairman: July 31 1972

ACKNOWLEDGMENTS

I would first like to thank Professor Vladimir Slamecka, whose interest and assistance during my graduate career often went beyond the call of duty and who helped make my stay at Georgia Tech so profitable.

The members of my guidance and reading committees, Professors Grosky, Gwynn and Kelly, deserve my thanks for their helpful criticism and encouragement during my research. In addition, John Gehl and Professor James Gough aided in the proof-reading of two drafts of this thesis.

I would also like to thank Professor Bas C. van Fraassen of the University of Toronto for reading and commenting on two drafts of this dissertation. It was Professor van Fraassen's excellent textbook on formal semantics which inspired the point of view presented here.

The Los Alamos Scientific Laboratory generously aided me during two summer research appointments. Most of the work reported here was sponsored by NSF Grant GN-655. This support is gratefully acknowledged.

My thesis advisor, Professor Lucio Chiaraviglio, has been of inestimable help to me. Not only are his many contributions on logical matters evidenced here, but his work as a teacher and a scholar and his friendship have left a lasting impression on me.

Finally, I am most indebted to my wife, Diane. Her patience, encouragement and aid during three years of graduate school could never be repaid, and were appreciated more than she ever knew. I also thank her for her expert typing of this dissertation.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

Semantic Theories in Programming

Languages of command provide important components of study in logic, mathematics and the computer sciences. To the logician, a command language offers a radical departure from the classical and near-classical systems of logic and therefore a challenge in its formalization. The central question is then ([44], p. 3):

> Can one articulate appropriate concepts of inference and of validity in such a way as to legitimate the inference of a command conclusion from premisses consisting of other commands . . . this question of . . . 'valid inference' among commands is . . . the key problem of the logical theory of commands.

The mathematician, on the other hand, is not solely concerned with the formal question of inference. Mathematical arguments frequently involve algorithmic constructions (this is not meant to imply that all such constructions can be realized effectively or recursively, but merely that a comprehensible and formal set of rules is provided), and mathematical practice dictates that expressions involving these constructions be correct insofar as they lead to some intuitively desired result. There are, in fact, rather large portions of mathematics which are involved in the construction of algorithms (e.g., numerical methods) and with certain properties (e.g., convergence) of the processes which they specify.

Finally, computer scientists have at the foundation of their discipline concrete examples of the sorts of languages described above. Languages which could be used to program computing machines have only

recently been utilized in standard mathematical and logical methodologies (as, for example, in Platek's application to recursion theory [41] and Engeler's theory of geometrical constructions [11]). It is clear, however, that these languages are the appropriate ones for the tasks of formalizing the logic of commands and mathematical constructions, since they all specify processes with the following characteristics:

(1) A state of a universe is given as an initial state.

(2) A question is asked about the current state of the universe.

(3) If the question is answered positively, then a procedure t^+ is invoked and the process proceeds from (2) with a new question and current state. Similarly, if the question is answered negatively a procedure t^- is invoked and control is directed back to (2).

Programs then have a mathematical meaning [46] when the language-objects called procedures are interpreted in some inductive fashion as transforming mathematical objects. On the other hand programs have a metamathematical meaning [10] when they are interpreted as experiments to be carried out in a given universe. Experiments of this sort we will call processes for reasons which will become apparent.

For us, a programming language PL is a logical language which formalizes the steps given in (1)-(3). In another paper [6] we discuss some methodological implications of these views. In this dissertation we restrict our attention to specifications of processes to be carried out in set theoretical universes. Let us make a technical distinction concerning the syntax of PL. The grammatical syntax of PL consists of a list of objects primitive to PL and a list of rules for combining these objects into various kinds of well-formed expressions. The primitive

objects of PL are:

(1) a denumerable set $\{\varphi_0,\ldots,\varphi_n,\ldots,\psi_0,\ldots,\psi_n,\ldots\}$ of conditions;

(2) a denumerable set $\{t_0,\ldots,t_n,\ldots\}$ of procedures;

(3) a denumerable set $\{\ell_0,\ldots,\ell_n,\ldots\}$ of labels;

(4) the symbols 'if', 'then', 'else', 'do', 'go to';

(5) the symbols ';', ':', '[', ']'.†

An instruction of PL is a string of symbols

$$\underline{\text{if}}\ A\ \underline{\text{then}}\ [\underline{\text{do}}\ P^+;\ \underline{\text{go to}}\ L^+]\ \underline{\text{else}}\ [\underline{\text{do}}\ P^-;\ \underline{\text{go to}}\ L^-]$$

where A is a condition, P^+ and P^- are procedures and L^+ and L^- are labels. If INST is an instruction of PL, then for any label L,

$$L:\underline{\text{INST}};$$

is a labeled instruction. A program is a finite collection of distinctly labeled instructions.

The logical syntax of PL consists of the grammatical syntax to which has been appended a designation of certain primitive objects or classes of primitive objects as logical constants, together with a set of rules which allow transformations of expressions that preserve some essential properties of the logical constants appearing in the expressions.

Grammatical syntax for programming languages has received intensive study in formal language theory. These results are of only passing interest here and, if necessary, problems can be avoided by passage to an abstract syntax for PL [33].

---

†In the sequel we will not make the distinction between use and mention of symbols.

Considering the effort expended in formal language theory, the logical syntax of programming languages is an amazingly neglected topic. We take the point of view [49] that " . . . the aim of a logical [syntax] is to provide us with a syntactic characterization of semantic notions . . . " For this reason it is fortunate that programmers and compiler writers have strong, although intuitive, notions of what their linguistic constructs are intended to mean.

The purpose of this dissertation is to utilize this generally accepted intended interpretation of programming languages and some standard techniques from the general semantic theory of first order logic to propose a way in which to view the semantics of programming languages. The point of view proposed herein would be suspect if it did not offer the hope of formalizing (in the logical syntax of PL) its most basic notions, so we also apply the methods to a class of simple programming languages to obtain some results concerning a rudimentary logical syntax for them. We, in fact, obtain a version of Rescher's "appropriate concepts of inference and validity" for the languages we choose to examine. These turn out to be roughly parallel to the corresponding notions in propositional logic. We also sketch some explicit connections with the theories of abstract computers and give indications that the results of this dissertation are in fact applicable to the semantic analysis of more familiar programming languages.

Aside from the intrinsic interest in formalizing languages of command, the semantics of programming languages has drawn the interest of computer scientists for a number of reasons:

(1) a formal, machine-independent theory of description is widely

held to be the appropriate manner in which to pass from language definitions to language processors;

(2) the thorough understanding of language defined invariants in an "execution time" environment has both pedagogic value and applications to various problems in compiler construction, in compiler optimization, and in parallel computation;

(3) proofs of correctness depend heavily on the interpretability of logical expressions in the same universes in which the programs are to be executed;

(4) projects in program synthesis and verification need secure foundations to study how more complex behavior may be "inferred" from certain regular ways of combining program segments;

(5) many problems in and applications of "pure recursion theory" rest on bases similar to the ones described above (for example, Platek's dissertation [41]; also, extensions of the Blum measures and other measures of complexity can be construed to depend not only on the absolute -- recursive -- power of the program, but on the difficulty of interpreting a program as specifying a process which has a given property).

Historical Sketch

The semantic theory of programming languages received idiosyncratic treatment in the ALGOL Report [38]. Proposals leading to theories aimed at defining a "processor-independent" semantics appear in early papers by I. I. Ianov [15], R. W. Floyd [13], and J. McCarthy [30,31].

In [13] Floyd outlined a method for associating with various

statement types appearing in a program (flow-diagram) certain first order formulas with free variables ranging over states of a universe.

Z. Manna [25,26] extended these proposals to a number of algorithms which describe the construction of W-formulas for programs and program schemes. W-formulas relate to conditions in given programs by properties of logical correctness. In [26], for instance, an effective procedure was given for arriving at a W-formula W_p for the program p such that p terminates just in case $\sim W_p$ is valid. Certain problems in the correctness and equivalence of programs have also been investigated from this basis. R. London [23] has applied these and similar methods to obtain proofs of correctness.

McCarthy made a number of explicit proposals concerning the desirability of paralleling model-theoretic methods. The early results [30,31] made use of conditional expressions in proofs of equivalence. Conditional expressions have since been incorporated into a number of studies of program schemes [40]. Later development of McCarthy's method of recursion induction has resulted in a number of induction principles: truncation, fixed point and structural induction. Floyd's methods are sometimes collectively referred to as the method of inductive assertions. These and related methods are surveyed by Manna, Ness and Vuillemin [27]. Structural induction appeared in a subsequent paper by McCarthy and J. Painter [34], proving the correctness of a compiler for arithmetic expressions. McCarthy also introduced [33] abstract syntax as a solution to the general description problem for languages by language components and their properties.

The description problems were seriously tackled by P. Lucas,

K. Walk and others [24] in the description of PL/1, using a form of abstract syntax called Vienna Definition Language. Language features (including block structuring with declarations and scope identifiers, procedure definitions, recursive calls, parameter call-by-name and call-by-value) have also been recovered in a variety of lambda-calculus and combinatory logic models. Important systems of this type have been proposed by P. Landin [21], C. Strachey [48], J. Morris [37] and R. Orgass [39].

The axiomatics of description processes has been examined from a number of points of view. C. A. R. Hoare [14] related an input property P_I of states to an output property P_0 of states by "passing through" program segments whose actions have been defined axiomatically. J. W. DeBakker used a more formal sense of axiomatics [4] to clarify certain properties of assignments and conditionals. The completeness of a formal system sufficient to describe McCarthy's assignment function was first demonstrated by D. Kaplan [19]. P. Lauer [22] proposed in his dissertation a first order theory of interpretation for programs. The theory was proved consistent by constructing (essentially) an abstract computer IM which correctly interpreted programs (IM, a model of the theory). S. Igarashi's early approaches [16] to equivalence were also axiomatic in character.

Algebraic topics (usually from category theory) have been investigated by Igarashi [17], C. C. Elgot [8] and others. Related work includes that of D. Benson [2] and D. Knuth [20].

Program synthesis and verification have been considered as related to some of this work in [12,28,3]. Synthesis approaches are proof-

theoretic in format and lean heavily on the induction principles mentioned above to implement looping. Verification is to some extent available in London's method [23], although progress in automation has been slight.

Ianov's original work with uninterpreted program components [15] gave rise to an active area of research in program schemes and models of parallel computation. In addition to much of the work cited above, specific linear schemes, flow graphs and matrices have been examined extensively, their equivalence problems investigated and their relationship to automata and decidability discussed.

D. Scott [47] and E. Engeler [10] have each constructed bona fide models of programming languages. Scott's approach was algebraic and syntactic, while Engeler's was metamathematical and semantic. Scott constructed an algebra of syntax (a complete lattice based on the ordering "is less defined than") for a portion of a programming language represented geometrically. Engeler has given an effective procedure which associates with each program p an infinitary formula φ_p of $L_{\omega_1\omega}$ so that when p is interpreted in $\mathfrak{B}$ and terminates in $\mathfrak{B}$, $\mathfrak{B} \models \varphi_p$. The auxiliary notion of algorithmic basis was applied to the theory of geometrical constructions in [11].

Algebras and Abstract Computers

The algebraic theory of abstract computers was elaborated by J. Poore [42] and applied to automata theory by R. Roehrkasse [45]. We will sketch here only the barest outlines of the extant theory.

An abstract computer is an algebra which can be viewed as a model

of a first order theory of programmable, centrally controlled, iterative, synchronous, non-interactive, discrete parameter computing machines (the adjectives are Roehrkasse's). In its simplest version, an abstract computer is a triple $\langle S,A,C\rangle$ where S is a nonempty set of states, A is a nonempty set of mappings $S\to S$ and $C:S\to A$. Thus, the sequence

$$\langle s,\ (C(s))(s),\ (C(C(s))(s))((C(s))(s)),\ldots\rangle \tag{1}$$

defines a sequence of states of the computer which depends only on the choice of the initial state s. We can define a total state transition function $T:S\to S$ by

$$T(s) = (C(s))(s).$$

Hence, (1) is more naturally written

$$\langle s,\ T(s),\ T^2(s),\ldots\rangle$$

and we obtain an alternative formulation of $\langle S,A,C\rangle$ by considering $\langle S,T\rangle$. If $\langle S,T\rangle$ is an abstract computer, each sequence

$$\pi_T = \langle \pi_T(\alpha) : \alpha<\omega \text{ and } \pi_T(\alpha) = T^{\alpha}(\pi_T(0))\rangle$$

is a process of $\langle S,T\rangle$. Usually, π_T is written π. Let π be a process of $\langle S,T\rangle$. If for some $\alpha<\omega$ we have

$$\pi(\alpha+1) = \pi(\alpha),$$

we say that π terminates after α iterations just in case α is the least such ordinal. Clearly, if π terminates after α iterations, then for all $\beta\geq\alpha$,

$$\pi(\beta) = \pi(\alpha).$$

In general, we refer to $\pi(\alpha)$ as the value of the process after α iterations.

Since each choice of $s \in S$ determines a unique process, we can view the activity of programming $\langle S,T\rangle$ to generate the process π as selecting the initial state $\pi(0)$. This is something quite different from specifying this selection, which is the subject of this dissertation.

There are several useful restrictions which can be placed on the definition of abstract computers. Let $\langle S,A,C\rangle$ be an abstract computer. If for some $X \neq \phi \neq Y$, we have

$$S = \{f: f:X \to Y\}$$

and for each $s \in S$,

$$\{x:(C(s)(s))(x) \neq s(x)\}$$

is a finite set, then $\langle S,A,C\rangle$ is said to be a finitary action computer. We can obtain a refinement of finitary action computers as follows. Let $\langle S,A,C\rangle$ be as above. If, in addition, S is a Boolean algebra with dual space S^* and if for any $s \in S$ and $a \in A$ there is a finite $K \subseteq S^*$ such that

$$(a(s)+s)(k) = 0, \quad k \in S^*-K,$$

then $\langle S,A,C\rangle$ is an abstract digital computer.

Homomorphisms and isomorphisms are morphisms of the algebras. If $\langle S,A,C\rangle$ is a Boolean algebra with operators we may also require that morphisms be Boolean morphisms.

The following representation theorem is due to Poore [42]. Let 0 be the simple Boolean algebra and for each nonempty X, let 0^X be the set of all mappings X→0. Define for each subset J of X the mappings R(J) and S(J) so that each maps $0^X \to 0^X$ and:

$$(R(J)(f))(x) = \begin{cases} 0, & \text{if } x \in J \\ f(x), & \text{otherwise} \end{cases}$$

$$(S(J)(f))(x) = \begin{cases} 1, & \text{if } x \in J \\ f(x), & \text{otherwise} \end{cases}$$

Let RS be the closure under functional composition of the set of all such mappings. Then every abstract digital computer <B,A,C> is isomorphic to an abstract digital computer with total state transition function defined in terms of RS only. Indeed, by letting $b \in B$ and

$$C^*(b) = S(((C(b)(b) \wedge b')^{-1}(1))R(((C(b)(b)' \wedge b)^{-1}(1))$$

we get

$$C^*(b)(b) = C(b)(b)$$

whenever $b \in B$. But we know that B is isomorphic to a subalgebra of 0^{B^*}.

It is much easier for most purposes to work with <S,T> rather than with <S,A,C>. Whenever we refer to an abstract computer, we actually mean the abstract computer <S,T> obtained from <S,A,C>.

Plan of Presentation

In Chapter II we present the basic constructs of the semantic systems in which we will be working. A version of the standard methods for

evaluating nonlogical constants of first order theories (which we call languages) is presented and applied to motivate the formal notion of state of a universe. Procedures are explicated by means of first order properties and a corresponding definition of program is introduced. The remainder of Chapter II is devoted to familiarizing the reader with the semantic system which results when programs are interpreted as processes.

Chapter III begins with a parallel construction of logical systems for programs and for logical languages with a single binary connective. Success properties of programs are then explicated as the general truth paradigm for programs. It is shown that many semantic properties of first order languages are passed along to the programming languages which are constructed from them. A main result of Chapter III is the theorem which asserts the existence of a unary logical connective functioning as exclusion negation [49]. The Boolean nature of programs follows from this result, and this allows us to place the algebraic structure of programs in the familiar setting of algebraic logic [43] (the so-called Lindenbaum-Tarski algebras).

We conclude Chapter III with a number of corollary results. Namely, we show: that even in a radical expansion of the command structure of our programming language, the Boolean properties are preserved; that some previously vital restrictions on the methods for evaluating programs can be relaxed; and that the algebraic point of view has application to the theories of abstract computers.

Finally, in Chapter IV, we resolve a conceptual difficulty involving execution time complexities of programs, summarize the results, and present a number of suggestions for future research in the setting put

forth in this dissertation. A bibliography of selected references in the semantics of programming languages and proving properties of programs follows Chapter IV.

CHAPTER II

PROGRAMS AND THEIR INTERPRETATION

Preliminaries

Throughout most of what follows, we will be working in a set theory with classes and the Axiom of Choice. The existence of proper classes must be guaranteed since the model theory of first order languages commits us to the discussion of very large collections of relational structures, and much of our theory derives from model theory. This requirement is, however, purely a technical one. Once we have assumed it there will be no need to explicitly distinguish sets and classes. Thus, in particular, we will allow ourselves considerable latitude in the construction of expressions involving proper classes, with the understanding that all of the arguments of interest could be carried out in the set theory using only sets.

Cartesian powers of sets, denoted A^B for nonempty A and B, are taken to be sets of all mappings $f:B\to A$. Although we will let 2^A be the power set of A, A^n be the n-fold Cartesian product of A with itself, and occasionally refer to powers of cardinal and ordinal numbers, we do not anticipate any confusion over the intended meaning.

An ordinal is the set of all smaller ordinals. Cardinals are initial ordinals. Thus, for ordinals α,β we use $\alpha\in\beta$ and $\alpha<\beta$ interchangeably. Ord(A) is the ordinal of A and Card(A) its power. I is the collection of of all finite ordinals and is usually taken to be the natural numbers

$\{0,1,2,\ldots,n,\ldots\}$ so that $I = \omega$.

If $\{A_i\}_{i\leq n}$ is an indexed collection of sets and

$$a = \langle a_1,\ldots,a_n\rangle \in \underset{i\leq n}{\times} A_i,$$

we write ka for a_k whenever $1\leq k\leq n$.

By a language L_μ we mean a fragment of a first order predicate calculus with equality to which we append a (possibly empty) set of non-logical axioms. The <u>type</u> μ of L_μ is a pair of functions $\langle\mu_1,\mu_2\rangle$ such that $\mu_1\in\omega^\alpha$, $\mu_2\in\omega^\beta$ and:

(1) for each function symbol f_γ of L_μ, f_γ is a $\mu_1(\gamma)$-ary function symbol;

(2) for each predicate symbol q_γ of L_μ, q_γ is a $\mu_2(\gamma)$-ary predicate symbol;

(3) each individual constant is a function symbol f_γ with $\mu_1(\gamma) = 0$.

If there is no possibility of confusing types, L_μ will simply be denoted by L. L must contain a denumerable supply of variables.

Since programming languages do not typically contain conditional statements of the form

$$\underline{\text{if}}\ (Qx_i)\varphi(x_i)\ \underline{\text{then}}\ldots\underline{\text{else}}\ldots,$$

where Q is a quantifier, we will need to speak of the language L(QF) which is obtainable from L by the well-known methods of eliminating quantifiers through introduction of Skolem functions. In a number of places (see, e.g., [43]) this has been shown to be an inconsequential modification of L. Unless noted otherwise, L(QF) is to be viewed as the set of

quantifier-free formulas obtainable from L. We will not distinguish the types of L and L(QF).

We will denote by T the set of terms of L(QF). Every term is a procedure in the sense described in the previous chapter. T contains a nonempty set of terms called e-terms (e-procedures). For simplicity, we let the symbol e denote any e-term. e-terms will be defined precisely semantically later in this chapter.

A relational structure (or, simply, a structure) is a triple $\mathfrak{B} = \langle B,F,R\rangle$, where B is a nonempty set called the universe of $\mathfrak{B}$, $F = \langle f_\gamma : \gamma<\omega \text{ and } f_\gamma : B^n \to B,\ n>0\rangle$ and $R = \langle P_\gamma : \gamma<\omega \text{ and } P_\gamma \subseteq B^m,\ m>0\rangle$. We also permit individuals in F as nullary functions. A type of structures is defined as before:

(1) for each $g_\gamma \in F$, $g_\gamma : B^{\mu_1(\gamma)} \to B$;

(2) for each $P_\gamma \in R$, $P_\gamma \subseteq B^{\mu_2(\gamma)}$;

(3) if $g_\gamma \in B$ and $g_\gamma \in F$, then $\mu_1(\gamma) = 0$.

We realize L in $\mathfrak{B}$ as follows. Let $\mathfrak{B}$ be of type μ. Then:

(1) if f_γ is a function symbol of L, then

$$\underline{f}_\gamma : B^{\mu_1(\gamma)} \to B$$

is in F;

(2) if q_γ is a predicate symbol of L, then

$$\underline{q}_\gamma \subseteq B^{\mu_2(\gamma)}$$

is in R;

(3) if f_γ is an individual constant, then

$$\underset{\gamma}{f} \in B$$

is in F.

The class STR (actually, STR_μ) is the class of realizations of L_μ if and only if L_μ is realized in each $\mathfrak{B} \in STR$ and for each nonlogical axiom φ of L_μ, $\mathfrak{B} \models \varphi$, which is to be read, "$\varphi$ is true in $\mathfrak{B}$." (For a complete discussion of truth by satisfaction in a structure, see [1].) If L_μ contains nonlogical axioms and $\mathfrak{B} \in STR$, then we sometimes say that $\mathfrak{B}$ is a model of L_μ, and if for some formula φ of L_μ $\mathfrak{B} \models \varphi$, then $\mathfrak{B}$ is a model of φ.

If $\mathfrak{B}$ and $\mathfrak{B}'$ are isomorphic structures, we write $\mathfrak{B} \cong \mathfrak{B}'$. If $\mathfrak{B}$ and $\mathfrak{B}'$ are elementarily equivalent, we write $\mathfrak{B} \equiv \mathfrak{B}'$.

For any language L, not necessarily restricted to the usual first order logic, its Lindenbaum-Tarski algebra is the atomless lattice (usually a Boolean algebra) of equivalence classes of formulas of L, denoted LTA(L), which is isomorphic to the set algebra of elementary classes of L over STR. For any lattice B, $\Delta(\nabla)$ denotes an ideal in B (filter in B). If B is a Boolean algebra and ∇ is a maximal proper filter in B, it is an ultrafilter.

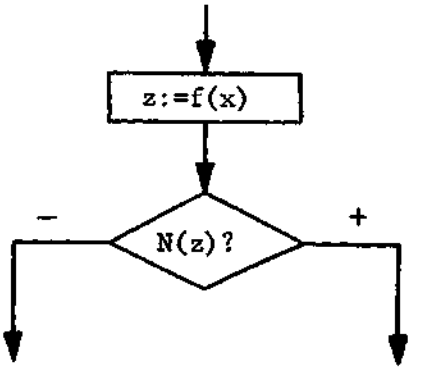
In the sequel, any notation not introduced above will be followed by a reference to a source of standard use.

States of a Universe

As we remarked above, a program describes a set of transformations to be carried out on a state of a universe. By a state of a universe B we now mean a sequence of elements of B. The definition we intend to adopt recalls devices proposed by McCarthy [32,35] and used by Lauer [22] in his consistency proofs for a first order semantic theory and by Kaplan

[19] in his completeness proofs for a formal system of assignment commands.

Suppose that a universe B (i.e., relative to a relization $\mathfrak{B}$ of L(QF)) has been given and a number of transformations recorded as specified by a program p. If p then calls for a conditional branch to be made, depending on the satisfaction of some formula $\varphi \epsilon L(QF)$ in $\mathfrak{B}$, we have the following intuitive result. Since φ is only to be tested against those states which are in the range of the specified transformations, φ can be true as far as the program is concerned without having $\mathfrak{B} \models \varphi$. That is, sufficient transformation of states of B may have been effected to insure that whenever φ in p is to be tested against a state of the universe passed on to it from a preceding control point in the program, a set theoretic truth of $\mathfrak{B}$ will result. For example, suppose that $f(x) \epsilon T$ is realized in $\mathfrak{B}$ as the integer part of any real x, that z = f(x) and that the open formula N(x) is realized as "x is an integer." Consider the program segment represented by the flow diagram shown below:



A test on N(z) will always result in a positive branch when N(z) is tested against a state of B in the range of the transformation realized from the command

z:=f(x);

although, in general, there is no reason to suppose that

$$\mathfrak{B} \models N(z)$$

holds.

We will say that a state of the universe B in $\mathfrak{B} = \langle B,F,R\rangle$ is a mapping $\bar{b}:T\to B$, defined so that if $t\in T$ contains no free occurrence of an individual variable of L, then:

(1) if t is an individual constant

$$\bar{b}(t) = \underline{t};$$

(2) if t is a term $f(t_1,\ldots,t_n)$

$$\bar{b}(t) = \underline{f}(\bar{b}(t_1),\ldots,\bar{b}(t_n)).$$

Since t contains no free variables, the definition is unambiguous. There is, however, some redundancy here; this will be cleared up in the next section. For convenience of notation we will write $\bar{b}\in B^T$ to indicate that $\bar{b}$ is a state of the universe B.

Thus, each term t of L(QF), for each state $\bar{b}$, names an element $\bar{b}(t)\in B$. It will be convenient to assume that T is ordered and that $Ord(T) = \omega + \omega$; specifically,

$$T = \langle x_0,x_1,\ldots,x_n,\ldots,t_0,t_1,\ldots,t_n,\ldots\rangle,$$

where for $i\geq 0$ each x_i is an individual variable, each t_i is a term distinct from an individual variable and

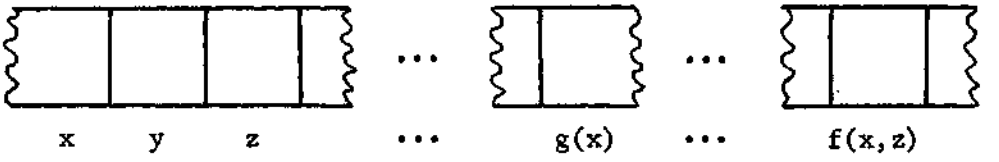
$$T = \{x_i : i\geq 0\}\cup\{t_i : i\geq 0\}.$$

We have also found it convenient to identify procedures with elements of T. The intended interpretation for these, the simplest procedures which can be specified by L(QF), is that when a term $f(t_1,t_2)$ appears in a program in the context

$$\underline{do}\ f(t_1,t_2)$$

the transformation induced, when applied to a state $\bar{b}$, replaces the position named by $f(t_1,t_2)$ with its realization. It will be argued later that this is in fact a special case of a more general notion of procedure.

Intuitively speaking, each $\bar{b} \in B^T$ is the linearization of the memory of some abstract device, as depicted below:



The "registers" are named by terms of L(QF). Some of these registers are sufficiently named by variables (e.g., identifiers, address names) and constants (e.g., programmable constants, literals, bit patterns), while others require more elaborate naming procedures. We may visualize the situation as follows. Consider a device D. D is constructed with "memory cells" addressed by identifiers and with "arithmetic registers" addressed by arithmetic expressions. Thus instead of isolating an accumulator, for instance, in which

$$\mathrm{ADD}(\ell_1,\ell_2)$$

will deposit the representation for

$$\text{CONTENTS}(\ell_1) + \text{CONTENTS}(\ell_2),$$

D has for each choice of ℓ_1 and ℓ_2 a register

$\text{ADD}(\ell_1,\ell_2)$

where the sum may be found only after $\text{ADD}(\ell_1,\ell_2)$ has been invoked as a procedure. Furthermore, it follows that after $\text{ADD}(\ell_1,\ell_2)$ has been called as a procedure, any further calls to it are superfluous, since

$$\bar{b}(\text{ADD}(\ell_1,\ell_2)) = \text{CONTENTS}(\ell_1) + \text{CONTENTS}(\ell_2)$$

before the second call. At other times we do not know that anything useful will be contained in the $\text{ADD}(\ell_1,\ell_2)$ register.

The objection might be raised that commands of actual programming languages seldom appear in this form -- that, for instance, x:=f(x) could hardly be construed as a term of any familiar first order language! This objection is treated in some detail in Chapter III, and by our assumptions on procedures nothing in the present development depends on such questions. It is pointed out in [22] that such "purely procedural" languages were outlined by D. Scott (September, 1969: Working Group WG2.2 on Formal Language Description Languages, Essex University, England)† as a basis of semantic theory. They are essentially languages without

†Also in private communication with Scott, March, 1972.

assignment, blocks and parameters.† The notion of state of a universe generalizes the notion of state of computation described in [22], although the formulations were obtained independently (see also "Historical Sketch" in Chapter I).

We will for the present assume that if an explicit assignment command appears as a procedure, its realization in

$$\underline{\text{do}}\ x_i := t$$

for $t \in T$ is a transformation which for each $\bar{b} \in B^T$ sends $\bar{b}(x_i)$ into $\bar{b}(t)$, leaving the rest of $\bar{b}$ unchanged.

Procedures

In this section we will describe a method for realizing terms and procedures in elements of STR. The method we propose here will be shown to be an inconsequential modification of the usual first order methods. The value of a term $t \in T$ at a state $\bar{b} \in B^T$, denoted $\underline{\bar{t}}(\bar{b})$, is obtained inductively:

(1) if t is an individual constant, then $\underline{\bar{t}}(\bar{b}) = \underline{t}$;

(2) if t is an individual variable, then $\underline{\bar{t}}(\bar{b}) = \bar{b}(t)$;

(3) if t is a term $f(t_1,\ldots,t_n)$, then

$$\underline{\bar{t}}(\bar{b}) = \underline{f}(\underline{\bar{t}_1}(\bar{b}),\ldots,\underline{\bar{t}_n}(\bar{b})).$$

Thus, for any atomic term t, $\underline{\bar{t}}(\bar{b}) = \bar{b}(t)$.

The semigroup of procedures, M_T, for L(QF) is the semigroup of

†In particular, none of our procedures introduce side-effects. Thus, the call of a procedure t at some point in the execution of a program will modify only those aspects of a state which are explicit in the procedure declaration.

finite length words over T generated subject to

$$et = te = t$$

for all $t \in T$. Elements of M_T are then "histories" of the commands invoked by particular programs. (Another valuable view of M_T is that elements of M_T are input tapes to an automaton -- hence, the definition of M_T as a semigroup! This way of looking at things will help us to achieve a decidability result in the next chapter.)

Elements of M_T are realized in STR in a natural way. For each $\mathfrak{B} \in STR$, $\|\cdot\| : M_T \to (B^T)^{B^T}$ is a homomorphism of semigroups defined so that if t is a term; i.e., if

$$\|t\| : B^T \to B^T$$

then

$$(\|t\|(\bar{b}))(t_0) = \begin{cases} \overline{t_0}(\bar{b}), & \text{if } t_0 = t \\ \bar{b}(t_0), & \text{otherwise} \end{cases}$$

for all $\bar{b} \in B^T$.

We are now in a position to define <u>e-procedures</u>. (This is not circular. We could have defined $\|\cdot\|$ for T only, then defined e-procedures, then M_T and finally the extension of $\|\cdot\|$ to the whole semigroup.) An element m of M_T is an e-procedure just in case

$$\|m\|(\bar{b}) = \bar{b}$$

for all states $\bar{b} \in B^T$, whenever B is the universe of a model $\mathfrak{B}$ of L(QF). Hence, every individual variable and every individual constant is an e-term and hence an e-procedure. This level of generality (at the semantic portion of PL) is necessary to the description of e-procedures since we may want to introduce other sorts of procedures into PL without changing the way in which they are evaluated at states of a universe.

Since $\|\cdot\|$ is a homomorphism of semigroups, it follows that if $m = t_n \cdot t_{n-1} \cdot \ldots \cdot t_1 \in M_T$, then

$$\|m\|(\bar{b}) = \|t_n\| \cdot \|t_{n-1}\| \cdot \ldots \cdot \|t_1\|(\bar{b})$$

for $\bar{b}$, a state of B. Thus, each procedure is realized as a transformation of states (cf. comment regarding assignment at the end of the preceding section). Notice that each transformation so defined modifies the appropriate register. This view may be found in some form in [10,19,32, 35,47], although none of the authors view the method in these general terms.

With respect to the $\|\cdot\|$ operation, each element of M_T which is a generator for the whole semigroup is an idempotent. That is, $\|t \cdot t\| = \|t\|$ for all $t \in T$. Similarly, if $m = t_n \cdot \ldots \cdot t_j \cdot \ldots \cdot t_1$, then

$$\|t_j \cdot m\| = \|m\|.$$

This will bring about a unique finitude restriction on processes generated by programs.

The following construction will be of use to us. For each $\varphi \in L(QF)$, order the terms which appear in φ first by the number of symbols which appear in the term and then lexicographically. Let

$$m_\varphi = t_n \cdot t_{n-1} \cdot \ldots \cdot t_0,$$

where under the prescribed ordering

$$t_i < t_{i+1}.$$

Clearly, if φ contains only atomic terms or if φ is a sentence of L(QF), then $m_\varphi = e$.

We now introduce a formal version of the intuitive notion of "true in a program." For each $m \in M_T$, a formula φ of L(QF) is said to be m-satisfied by a state $\bar{b}$ in $\mathfrak{B} \in STR$, written $\mathfrak{B} \models \varphi[m,\bar{b}]$, just in case:

(1) φ is an atomic formula $t_1 = t_2$ and

$$\|m\|\bar{b}(t_1) = \|m\|\bar{b}(t_2);$$

(2) φ is an atomic formula $q(t_1,\ldots,t_n)$ and

$$\langle \|m\|\bar{b}(t_1),\ldots,\|m\|\bar{b}(t_n)\rangle \in \underline{q};$$

(3) φ is $(\psi_1 \vee \psi_2)$ and either $\mathfrak{B} \models \psi_1[m,\bar{b}]$ or $\mathfrak{B} \models \psi_2[m,\bar{b}]$, or both;

(4) φ is $(\sim\psi)$ and it is not the case that $\mathfrak{B} \models \psi[m,\bar{b}]$.

Lemma 1 follows immediately from the definitions of m-satisfaction and state of a universe.

Lemma 1. If φ is a sentence of L(QF) and $\mathfrak{B} \models \varphi[m,\bar{b}]$ for some $\bar{b} \in B^T$ and $m \in M_T$, then $\mathfrak{B} \models \varphi[e,\bar{b}]$ for all $\bar{b} \in B^T$.

T has been ordered so that its first ω places are occupied by all and only the distinct variables in T. In the usual definition of truth (see [1], p. 55ff.) each sequence $\langle b \rangle \in B^\omega$ is such that

$$\overline{\underline{x_i}}(\langle b \rangle) = \langle b \rangle(i),$$

so that again distinct positions in $\langle b \rangle$ correspond to distinct variables. By the denumerability of T, there is a one-one correspondence v between variables and ω such that for each $\bar{b} \in B^T$, there is a $\langle b \rangle \in B^\omega$ for which $\bar{b}(x_i) = \langle b \rangle(v(x_i))$, and conversely (v is called the natural correspondence). The following theorem, showing that m-satisfaction introduces no novelty to the usual definitions, then follows immediately.

Theorem 1. Let $\langle b \rangle \in B^\omega$ and v be the natural correspondence between finite ordinals and individual variables in T. Suppose for each $x \in T$ which appears free in φ we have

$$\langle b \rangle(v^{-1}(x)) = ||m_\varphi||\bar{b}(x)$$

for $\bar{b} \in B^T$. Then $\mathfrak{A} \models_{\langle b \rangle} \varphi$ if and only if $\mathfrak{A} \models \varphi[m_\varphi, \bar{b}]$.

Thus, from Lemma 1 and Theorem 1 we can see that the semantic predicate 'true' has not been significantly modified in the passage to m-satisfaction. Moreover, m-satisfaction is more easily carried into treatments of complex programming languages than is the more orthodox notion of satisfaction (see Chapter III).

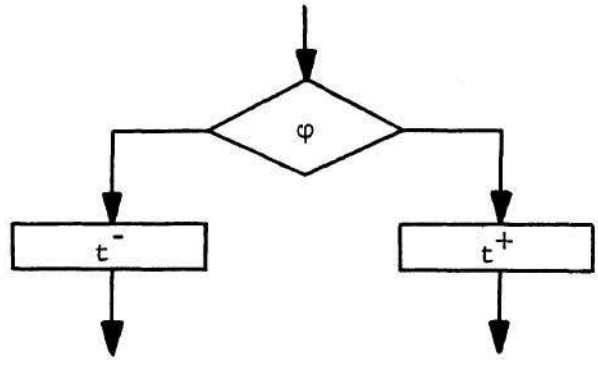
First Order Programs

Our most immediate goal is to examine the extent to which the (largely unknown) logical structure of a programming language can be reflected algebraically in terms of its interpretations. The definition of program we will use is a reflection of several versions of program schemes. The main novelty in this case, aside from our simplifying assumptions about procedures, is that we permit programs to be infinite in length. This appears startling until the following qualification is

made: each program can only be interpreted to be commanding significant actions at a finite segment. That is, infinite programs are interesting only at a finite number of instructions.

Programs contain no distinguished start instructions. The argument for relaxing this restriction is a very simple one. To single out a starting place in a program is to decree that the execution of the program lead toward a goal predefined by the author of the program (e.g., to compute a specific recursive function). This is not properly a semantic notion but a pragmatic one. It is concerned with the relationship of the code to the code user, and is therefore not of fundamental interest here. A program which is intended to begin at line ℓ_0 might just as easily be construed as beginning at lines $\ell_1, \ell_2, \ldots, \ell_n$. In fact, we are not at all concerned with the single process which a program is intended to specify; rather, we are concerned with the conditions which processes generated by a program place on its interpretation. As Theorem 4 will show, however, no generality is lost in assuming that a class of processes be generated by programs from a distinguished line ℓ_i, since there is always a consistent relabeling of programs which does not affect their interpretation.

Finally, programs when exhibited in flow diagram form each have instructions of the sort shown below:



That is, each instruction in the programming language takes the form

$$\underline{\text{if}}\ \varphi\ \underline{\text{then}}\ldots\underline{\text{else}}\ldots.$$

A mapping $p:I\rightarrow(L(QF)\times T\times I\times T\times I)$ is a <u>program</u> of PL(QF) if and only if there is a finite $K\subset I$ such that for each $i\in I-K$

$$p(i) = (\varphi,e,i,e,i)$$

for some $\varphi\in L(QF)$. If p is a program, each $p(i) = (\varphi,t_1,j_1,t_2,j_2)$ is a <u>line</u> of p and is to be viewed as the labeled instruction

$$i:\ \underline{\text{if}}\ \varphi\ \underline{\text{then}}\ [\underline{\text{do}}\ t_1;\ \underline{\text{go to}}\ i_1]\ \underline{\text{else}}\ [\underline{\text{do}}\ t_2;\ \underline{\text{go to}}\ i_2];\ .$$

Elements of I are called <u>labels</u> of PL(QF). If $p(i) = (\varphi,e,i,e,i)$ for some $\varphi\in L(QF)$, we will write $p(i) = e^*$. Each label i at which $p(i) = e^*$ is called an <u>exit</u> of p just in case there is some $j\in I$ such that $p(j) \neq e^*$ and

$$i\in\{3p(j),5p(j)\}.$$

In general, if p is a program and K is the least subset of I for which $p(i) = e^*$ whenever $i\in I-K$, we will exhibit p explicitly by listing p(i) for each $i\in K\cup\{j:\ j \text{ is an exit of } p\}$ only.

Clearly there are programs distinct according to this definition which we would not intuitively call distinct. For instance, consider the programs p and p' which are related by

$$p'(i) = p(i)$$

for all $i\in I$ at which $p(i) \neq e^*$, and for all $i\in I$ such that $p(i) = e^*$,

$$p'(i) = e^*,$$

but

$$lp(i) \neq lp'(i).$$

Now, let us call two programs p and p′ equivalent when they have this property. That is, p is equivalent to p′ when p(i) = e* if and only if p′(i) = e* for all i∈I, and p and p′ are identical elsewhere. Certainly the programs p and p′ may be syntactically distinct programs, but notice that they are distinct only at those lines which do not really matter. Equivalent programs are distinct only at lines corresponding to inaccessible e* lines or exits. Henceforth, by a program we shall mean an equivalence class of programs under this relation. This restriction enables us to obtain a denumerable set of programs and to identify programs which would not be called distinct by programmers. While (new) programs are actually classes of (old) programs, we will continue to treat them as individual mappings. Unless noted otherwise, PL(QF) is the set of programs in the programming language PL(QF).

Lemma 2. Card(PL(QF)) = ω.

Valuations and Processes

Let $S = (PL(QF) \times M_T \times I)$. A valuation for PL(QF) is a mapping $r: S \to S$ such that for all $p \in PL(QF)$, $m \in M_T$, $i \in I$,

$$r(p,m,i) \in \{(p, 2p(i) \cdot m, 3p(i)), (p, 4p(i) \cdot m, 5p(i)\}.$$

If r is a valuation, $(p,m,i) \in S$ and $r(p,m,i) = (p, 2p(i) \cdot m, 3p(i))$ we will write r(p,m,i) = +; otherwise, we will write r(p,m,i) = -. We should

also consider the possibility that for valuations r_0 and r_1, $r_0(p,m,i) = +$ and $r_1(p,m,i) = -$ are indistinguishable. In general, we will not do this since the mechanisms for handling such cases will usually be clear from the immediate context and to add this complication just includes a trivial case in most proofs.

A valuation r is said to be <u>correct in $\mathfrak{B} \in STR$</u> just in case for all $(p.m,i) \in S$ we have

$$r(p,m,i) = +$$

if and only if

$$\mathfrak{B} \models lp(i)[m,\bar{b}], \text{ for all } \bar{b} \in B^T. \tag{2}$$

By a valuation we will usually mean a valuation correct in a model. Notice that the condition (2) is intuitively a necessary and sufficient condition, since correctness is defined only for valuations and by the definition of valuation if the 'true' branch is not taken, then the 'false' branch must be taken.

<u>Lemma 3</u>. For each valuation r, the pair (S,r) is an abstract computer.

The class PC is the class of abstract computers of the form (S,r), for r a valuation correct in a model of L(QF). The set of <u>processes</u> of PL(QF) is precisely the set of processes of PC. That is, each process is a sequence

$$\langle r^{\alpha}(p,m,i) : \alpha < \omega \rangle.$$

A process summarizes the external data independent behavior of a

program in an interpretation of L(QF). That is, r(p,m,i) = + just in case lp(i) is m-satisfied by <u>all</u> $\bar{b}$. In other words, we are interested here only in the programming language analogs of <u>sentences</u> of L(QF). That we can obtain all program behaviors of interest is obvious, since if a program depends on elements $b_0, b_1, \ldots, b_n$ of some universe B, we can by introducing sufficiently many individual constants into L(QF) create n programs p_k, $0 \leq k \leq n$, each of which assign the constants corresponding to the b_k explicitly <u>in the program</u>. This set of programs then represents the data dependent behavior of the original program. This is of course exactly the procedure which would be used to create a propositional language from L(QF).

<u>Lemma 4</u>. If $\mathfrak{B} \in$ STR, then there is a valuation r which is correct in $\mathfrak{B}$.

<u>Proof</u>. For each formula φ of L(QF) and each $m \in M_T$, either

$$\mathfrak{B} \models \varphi[m,\bar{b}] \text{ for all } \bar{b} \in B^T$$

or not. Thus, define r pointwise for each (p,m,i) with $lp(i) = \varphi$. Then r is a valuation correct in $\mathfrak{B}$.

<u>Theorem 2</u>. Let r_1, r_2 be valuations correct in $\mathfrak{B}_1$ and $\mathfrak{B}_2$ respectively. If $\mathfrak{B}_1 \cong \mathfrak{B}_2$, then $r_1 = r_2$.

<u>Proof</u>. Suppose there is some $s \in S$ such that $r_1(s) = s_1$, $r_2(s) = s_2$, but $s_1 \neq s_2$. Then for some $\varphi \in$ L(QF) we have $\mathfrak{B}_1 \models \varphi[m,\bar{b}_1]$ for all $\bar{b}_1 \in B_1^T$ but for some $\bar{b}_2 \in B_2^T$ not $\mathfrak{B}_2 \models \varphi[m,\bar{b}_2]$. Regardless of the syntactic form of φ, a contradiction arises. Let φ be the atomic formula $q(t_1,\ldots,t_n)$ and $K:\mathfrak{B}_1 \cong \mathfrak{B}_2$. Then for some $\bar{b}_2 \in B_2^T$,

$$<\|m\|\bar{b}_2(t_1),\ldots,\|m\|\bar{b}_2(t_n)> \notin \underline{q}$$

and hence

$$<K^{-1}(\|m\|\bar{b}_2(t_1)),\ldots,K^{-1}(\|m\|\bar{b}_2(t_n))> \notin K^{-1}(\underline{q}).$$

But for all $t\in T$

$$KK^{-1}(\|m\|\bar{b}_2(t)) = \|m\|\bar{b}_2(t),$$

which contradicts $\mathfrak{B}_1 \models \varphi[m,\bar{b}_1]$ for all $\bar{b}_1\in B_1^T$. The induction on the number of symbols in φ follows in similar fashion.

Thus, for instance, if L(QF) is categorical we have only to consider one valuation for PL(QF). For most applications however Theorem 2 implies that we will also have to consider more than this trivial case, since every incomplete language (e.g., Peano arithmetic) has non-elementarily equivalent models and hence non-isomorphic models.

Theorem 3. There are denumerably many sequences in S^{ω} which are processes of each $(S,r)\in PC$.

Proof. Let

$$p(0) = (\varphi,t,1,t,1)$$
$$p(1) = e^*$$

and

$$\pi(\alpha) = \begin{cases} (p,e,0), & \text{if } \alpha = 0 \\ (p,t,1), & \text{if } \alpha > 0. \end{cases}$$

As will be shown, the processes given in Theorem 3 serve as the "tautologies" of programming languages. The "meaning" of the programs

which generate the processes is essentially over-defined by PL. Suppose we equate processes just when they are processes of exactly the same computers in PC. Then when we identify two programs (actually, the processes which they generate) in this sense, we are doing something quite different than when we declare that the programs are equal as functions. In fact, the relation involved is more correctly read "if and only if"; that is, if processes π_1 and π_2 are processes of exactly the same computers, what we wish to convey by stating that

$$\pi_1 \equiv \pi_2$$

is not necessarily that π_1 and π_2 are responsible for computing the same function, but rather that the process π_1 is realizable if and only if the process π_2 is realizable. This is not the prevalent view of equivalence which appears in the literature [40] but from a logical point of view it is the more natural one. These notions will be further refined in the next chapter.

Theorem 4. Let p be a program and let $L_\sigma p$ be a program obtained from p by a transformation σ of labels. Then σ is a permutation of labels. Furthermore, for each process π with $1\pi(0) = p$, there is a process $L_\sigma \pi$ with $1L_\sigma \pi(0) = L_\sigma p$ such that π and $L_\sigma \pi$ are processes of exactly the same computers in PC.

Proof. For each $p \in PL(QF)$, let $\sigma: I \to I$ be such that if

$$p(i) = (\varphi, t_\cdot, j_\cdot, t_2, j_2)$$

then

$$L_\sigma p(\sigma i) = (\varphi, t_1, \sigma j_1, t_2, \sigma j_2).$$

Suppose σ is not one-one. Then there exist $i,j \in I$ for which $\sigma i = \sigma j = n$. Let k_i and k_j be elements of I which do not have this property with respect to σ, and let

$$p(i) = (\varphi_i, t_{1_i}, k_i, t_{2_i}, \ell_i)$$

$$p(j) = (\varphi_j, t_{1_j}, k_j, t_{2_j}, \ell_j).$$

Then $L_\sigma p(\sigma i) = L_\sigma p(n) \neq L_\sigma p(\sigma j) = L_\sigma p(n)$. This is clearly impossible. The equivalence of the processes is an immediate result of the definition of correctness of valuations.

The next theorem, showing that all processes of computers in PC eventually become periodic in their second members, is of fundamental importance to later developments.

Theorem 5. Let $\pi \in S^\omega$ be a process of some computer (S,r) in PC. Then there exists an $\alpha < \omega$ such that the sequence

$$\langle 2\pi(\beta) : \alpha \leq \beta < \omega \rangle$$

is periodic in period j for some $j \geq 0$.

Proof. The proof is by induction on the number of non-e* lines in a program. Let $\pi(0) = (p,m,i)$ and let p have ℓ non-e* lines. If $p(i) = e^*$ periodicity is immediate. If $\ell = 1$ and

$$p(i) = (\varphi, t_1, j_1, t_2, j_2)$$

then a simple combinatorial argument yields the following forms for π:

(1) $p(j_1) = e^*$ and either $3\pi(1) = j_1$ or $3\pi(2) = j_1$.

(2) $p(j_2) = e^*$ and either $3\pi(1) = j_2$ or $3\pi(2) = j_2$.

(3) Neither j_1 nor j_2 are exits. Then p loops on the ith line indefinitely.

In the first two cases, since π is terminating, it is periodic. In the last case we get periodicity from the equations:

$$\|t_1 \cdot t_1\| = \|t_1\|$$

$$\|t_2 \cdot t_2\| = \|t_2\|$$

$$\|t_1 \cdot t_2 \cdot t_1\| = \|t_2 \cdot t_1\| \tag{3}$$

$$\|t_2 \cdot t_1 \cdot t_2\| = \|t_1 \cdot t_2\|. \tag{4}$$

Suppose that the statement holds for all $\ell \leq k-1$. We divide p into three segments. The p_{k-1} part consists of all k-1 non-e* lines, p_k consists of the kth non-e* line and p_e consists of all e* lines of p. By the induction hypothesis, either p_{k-1} is periodic or there is a jump to p_k passing along the procedure $m \in M_T$. Denote p_k by the instruction $(\varphi, t_1, j_1, t_2, j_2)$. We only have to examine four possible iterations if there is a jump to p_k:

(1) $t_1 \cdot m$ is the procedure. If $p(j_1) = p_k$ this follows as in $\ell = 1$ above. If $p(j_1)$ is in p_{k-1}, then either the process is periodic or there is a jump to p_k.

(2) $t_2 \cdot m$ is the procedure. This is identical to (1).

(3) Either $t_2 \cdot t_1 \cdot m$ or $t_1 \cdot t_2 \cdot m$ is the procedure. This is also identical to (1).

(4) Either $t_1 \cdot t_2 \cdot t_1 \cdot m$ or $t_2 \cdot t_1 \cdot t_2 \cdot m$ is the procedure. The next

iteration begins the period, because of equations (3) and (4), respectively.

The next theorem deals with recovering given processes in arbitrary computers. Indeed, it turns out to be the case that extensional characteristics are related by straightforward algebraic techniques.

Theorem 6. Let π be a process of $(S,r)\in PC$. For any $(S,r')\in PC$, there are denumerably many processes π' of (S,r') such that $2\pi(\alpha) = 2\pi'(\alpha)$ for all $\alpha\geq 0$.

Proof. Let π become periodic with period β after α iterations in the sense described in Theorem 5. We will construct a program p' from which π' may be obtained. We denote by $t_{\alpha+1}$ the term t for which $t \cdot 2\pi(\alpha) = 2\pi(\alpha+1)$.

Suppose $\gamma<\alpha+\beta$. Let p and $\bar{p}$ be such that

$$p(n) = (\varphi, t_{\gamma+1}, \gamma+1, t, \ell)$$

$$\bar{p}(n) = (\varphi, t, \ell, t_{\gamma+1}, \gamma+1)$$

for any $t\in T$ and $\ell\in I$. Then define:

$$p'(\gamma) = \begin{cases} p(n), & \text{if } r'(p, 2\pi(\gamma), n) = + \\ \bar{p}(n), & \text{otherwise.} \end{cases}$$

Suppose $\gamma = \alpha + \beta$. Let p and $\bar{p}$ be such that

$$p(n) = (\varphi, t_{\alpha}, \alpha, t, \ell)$$

$$\bar{p}(n) = (\varphi, t, \ell, t_{\alpha}, \alpha)$$

for any $t\in T$ and $\ell\in I$. Then define

$$p'(\gamma) = \begin{cases} p(n), & \text{if } r'(p,2\pi(\gamma),n) = + \\ \bar{p}(n), & \text{otherwise.} \end{cases}$$

For $\gamma > \alpha+\beta$ let $p'(\gamma) = e^*$. Consider the sequence π' such that

$$\pi'(\delta) = r'^{\delta}(p',2\pi(0),0)$$

for all $\delta<\omega$. By the construction of p' we have for all $\delta<\omega$

$$r'(\pi'(\delta)) = (p',2\pi(\delta+1),\gamma),$$

where $\gamma \in [3\pi'(\delta)+1,\alpha)$. Therefore

$$2\pi'(\delta) = 2\pi(\delta)$$

for all $\delta<\omega$, and there are denumerably many choices for each $p(n)$ and $\bar{p}(n)$.

Since we have been careful to disturb the model theory of L as little as possible, it is not surprising that PL has a Löwenheim-Skolem Theorem associated with it. We prove a downward version.

Theorem 7. If r is a valuation correct in any infinite model, then r is correct in a countable model.

Proof. Suppose r is correct in no countable model. Then there is a process π of (S,r) which is a process of no computer whose transition function is a valuation correct in a countable model. Note that there is a symmetric case for $r(\pi(\alpha)) = -$.

Let $\pi(\alpha) = (p,m,i)$ and $r(\pi(\alpha)) = +$. Thus, for any countable model $\mathfrak{B}$ we have for some $\bar{b} \in B^T$ not $\mathfrak{B} \models lp(i)[m,\bar{b}]$. Let

$$lp(i) = \varphi(x_1,x_2,\ldots,x_n)$$

be a formula with x_i free for each $1 \leq i \leq n$. Then there is a countable set of sentences

$$\Gamma = \{\varphi(c_{i_1}, c_{i_2}, \ldots, c_{i_n})\}$$

of L(QF) such that for each $\psi \in \Gamma$

$$\mathfrak{B} \models \sim\psi \text{ if and only if } \mathfrak{B} \models \varphi[\underline{c}_{i_1}, \underline{c}_{i_2}, \ldots, \underline{c}_{i_n}].$$

Hence, for any infinite cardinal β there is a model $\mathfrak{B}'$ of that power for which $\mathfrak{B}' \models \sim\psi$ for each $\psi \in \Gamma$ and therefore for some $\bar{b}' \in B'^{T}$ not $\mathfrak{B}' \models lp(i)[m, \bar{b}']$.

Suppose there is some infinite $\mathfrak{B}'$ for which $\mathfrak{B}' \models lp(i)[m, \bar{b}']$ for all $\bar{b}'$. Then for $\psi \in \Gamma$ we would have $\mathfrak{B}' \models \psi$ and hence $\mathfrak{B} \models \psi$ for some countable $\mathfrak{B}$. But this is a contradiction. Therefore $\mathfrak{B}' \models lp(i)[m, \bar{b}']$ for all $\bar{b}'$ holds for no infinite $\mathfrak{B}'$.

Thus, if (S,r) fails to generate a process π correctly in a countable model, then π is not correctly generated in any infinite model.

CHAPTER III

SEMANTIC PROPERTIES OF PROGRAMS

Systems and Properties of Processes

In logic the metamathematical predicate $\models$ stands between a structure $\mathfrak{B}$ and a formula φ in

$$\mathfrak{B} \models \varphi \tag{5}$$

just in case φ denotes a set theoretic truth of $\mathfrak{B}$. The general method employed is to define a relation $G \subseteq STR \times L$, so that the assertion

$$G(\mathfrak{B},\varphi)$$

is equivalent to (5).

In this section, we will parallel the construction of a semantic system based on such a relation between STR and PL. The actual construction provides no novelty over the original methods (see, e.g., [49]), but it does provide some limitative results, in the next section, concerning the nature of the logical connectives that we would expect to find in PL.

In this chapter, we denote $PL(QF) \times M_T$ by S^*. There is a shift in point of view here. Notice that $S = S^* \times I$. We will for the most part be concerned with relating syntactic characteristics (expressed in S^*) to processes generated from the programs of PL(QF) (sequences in S). By Theorem 4 all such processes can be supposed to begin at line 0 of a particular program. If not, there is a consistent relabeling of the

program from which an entirely equivalent process may be obtained.

We will now introduce a construction which will account for a logical connective between programs:

(1) If τ is a transformation on the set of variables of L(QF) and if p is a program in PL(QF), then $V_\tau p$ is a program identical to p in every respect except that $V_\tau p$ contains a free occurrence of the variable τx wherever x occurs free in p. Similarly, for any $m \in M_T$, m and $V_\tau m$ differ only in the transformed variables. A transformation τ such that $V_\tau p'$ and p ($V_\tau m'$ and m) share no variables is called <u>isolating</u> for p and p' (isolating for m and m').

(2) The set $K_p \subseteq I$ is said to <u>support</u> the program $p \in PL(QF)$ if:

$$K_p = (I-p^{-1}(e^*)) \cup \{i:(\exists j)(j \in (I-p^{-1}(e^*)) \wedge (i=3p(j) \vee i=5p(j)))\}.$$

A permutation σ of labels such that $L_\sigma p'$ and p share no supports is said to be <u>isolating</u> for p and p'.

(3) Let σ be an isolating permutation of labels for p and p'. Then for each $\varphi \in L(QF)$, the program $\hat{b}_\varphi$ is called a <u>bridge</u> program between p and $L_\sigma p'$ just in case

$$\hat{b}_\varphi(k) = (\varphi, e, \sigma 0, e, \sigma 0)$$

(where $\sigma 0$ is understood to be mapped <u>from</u> p') for every exit k of p with $\varphi \in L(QF)$ and elsewhere $\hat{b}_\varphi(k) = e^*$. We then construct a program p'' as follows:

$$p'' = (p;p')_{\sigma,\tau,\varphi}(i) = \begin{cases} p(i), \text{ if } i \in (I-p^{-1}(e^*) \\ L_\sigma V_\tau p'(i), \text{ if } i \in (I-(L_\sigma p')^{-1}(e^*)) \\ \hat{b}_\varphi(i), \text{ if } i \text{ is an exit of } p \\ e^*, \text{ otherwise} \end{cases}$$

where σ is an isolating permutation of labels, τ is a variable isolating transformation for p and p' and $\hat{b}_\varphi$ is a bridge program between p and $L_\sigma p'$.

This completes the construction of the program

$$(p;p')_{\sigma,\tau,\varphi}.$$

The semicolon operation is extended to elements of S* by the equation

$$((p,m);(p',m'))_{\sigma,\tau,\varphi} = ((p;p')_{\sigma,\tau,\varphi}, m \cdot V_\tau m).$$

By assuming some standard (canonical) naming procedures it is always possible to remove the dependence of $(s;s')$ for s, $s' \in S^*$ on σ, τ, and φ, so that $(s;s')$ is well defined for each choice of s and s'. Therefore, we only use the semicolon operator in this latter sense (it is clear that in this sense it is a well-defined operator).

We now introduce, informally, a property of processes which we will call a <u>success</u> property. A success property G is more properly associated with a program which generates the process, so that it is natural to write

$$\mathfrak{B} \rightarrow_G s, \; s \in S^*$$

if and only if r is correct in $\mathfrak{B}$ and

$$<r^{\alpha}(s,0):\alpha<\omega>$$

has the property G.

Since in this section our construction will not depend on the choice of G (or even on the fact that s is a program-procedure pair), let us write

$$\mathfrak{B} \rightarrow s$$

which is to be interpreted as "s succeeds in $\mathfrak{B}$." This, of course, refers to a specific notion of "success" relative to a process which is generated from s in $\mathfrak{B}$. As in Chapter II, we will be careful to choose a property G which treats programs as sentences.

For each $s \in S^*$, denote by H(s), $\{\mathfrak{B}:\mathfrak{B} \in STR \text{ and } \mathfrak{B} \rightarrow s\}$. For $X \in 2^{S^*}$, let

$$H(X) = \bigcap_{s \in X} H(s).$$

When G is explicit

$$H_G(s) = \{\mathfrak{B}:\mathfrak{B} \rightarrow_G s\}.$$

Any such function H must satisfy the following:

(1) Axiom. there exist s and s′ in S* such that

$$H(s) = STR \text{ and } H(s') = \phi;$$

(2) Axiom. for any s and s′ in S*

$$H(s;s') = H(s) \cap H(s').$$

Since H is ultimately defined in terms of some property G these axioms amount to restrictions on G.

For convenience in notation, we define the closure Cl(s) of each $s \in S^*$:

$$Cl(s) = \{s' : H(s) \subseteq H(s')\}.$$

The closure of a program-procedure pair Cl(p,m) might be considered to be the set of all $s \in S^*$ which (p,m) semantically entails. We will, in fact, use the notation

$$X \models s$$

for

$$s \in \{s' : H(X) \subseteq H(s')\} = Cl(X).$$

In logic, sets of sentences closed with respect to semantic entailment play a key role. Suppose $X \subseteq S^*$. Then X is called a system in S^* if and only if $X \models s$ implies $s \in X$. The class of systems in S^* is never empty.

Lemma 5. S^* is a system in S^*.

Proof. By Axiom (1), $H(s') = \phi$ and thus $H(S^*) = \phi$.

Lemma 6. (Van Fraassen, [49]) If X and Y are systems, then $X \subseteq Y$ if and only if $H(Y) \subseteq H(X)$.

Lemma 7. (Van Fraassen, [49]) If X and Y are systems, then:

(a) $X \cap Y$ is a system

(b) $H(X) \cup H(Y) \subseteq H(X \cap Y)$

(c) $X \cap Y = \{s: X \models s \text{ and } Y \models s\}$

(d) $X \cap Y = Cl(X \cap Y)$

(e) $X \cap Y = \{s: H(X \cap Y) \subseteq H(s)\}$

(f) $X \cap Y = \{s: H(X) \cup H(Y) \subseteq H(s)\}$

(g) $X = Cl(X)$

Lemma 8. There is a smallest system in S* with respect to the partial ordering $\subseteq$ on systems.

Proof. By Axiom (1), $H(s) = STR$ for some $s \in S^*$, so take $Cl(s)$.

We now define an operation + (called system union) on systems as follows. For $X, Y \subseteq S^*$, let $X + Y = Z$ if and only if:

(1) Z is a system in S*;

(2) $X \cup Y \subseteq Z$;

(3) for any system Z' such that $X \cup Y \subseteq Z'$, $Z' \subseteq Z$.

Lemma 9. (Van Fraassen, [49]) If X and Y are systems, then:

(a) $X + Y = Cl(X \cup Y) = \{s: H(X \cup Y) \subseteq H(s)\}$

(b) $H(X + Y) = H(X) \cap H(Y)$

Denote the collection of all systems in S* by SYST(S*) or when necessary, by $SYST_H(S^*)$ or $SYST_G(S^*)$.

Theorem 8. (Van Fraassen, [49]) $\langle SYST_H(S^*), \cap, + \rangle$ is a lattice.

Corollary. $\langle SYST_H(S^*), \cap, + \rangle$ is distributive.

Proof. The result follows by straightforward calculation from Lemmas 7 and 9.

Since SYST(S*) is at least distributive, it is interesting to ask whether or not there are additional lattice theoretic operations which will place SYST(S*) in a more familiar setting. If, for instance, the lattice turns out to be a Boolean algebra we can immediately infer some surprising results about the logical structure of programs. This will

not, in fact, be the case for arbitrary G (we will find a G for which the lattice of systems is the lattice of filters in a Lindenbaum-Tarski algebra), but we can obtain a more rigid structure on SYST(S*).

The pseudo-complement of X(mod Y), written $X \Rightarrow Y$, for X, Y systems in S* is some $Z \in SYST(S^*)$ such that Z is the greatest element for which

$$X \cap Z \subseteq Y$$

holds. If $X \Rightarrow Y$ exists for every X, Y in SYST(S*), then the lattice is said to be relatively pseudo-complemented.

Theorem 9. SYST(S*) is a relatively pseudo-complemented lattice.

Proof. Let $\Rightarrow$ be a binary operation on SYST(S*) such that

$$X \Rightarrow Y = \{s: H(Y)-H(X) \subseteq H(s)\}.$$

$X \Rightarrow Y$ is clearly a system for X, $Y \in SYST(S^*)$. By Lemma 7(b)

$$H(X) \cup H(X \Rightarrow Y) \subseteq H(X \cap (X \Rightarrow Y)).$$

Suppose $\beta \in H(X)$; then $\beta \notin H(Y)-H(X)$. Similarly, if $\beta \notin H(X)$, then $\beta \in H(Y)-H(X)$ if $\beta \in H(Y)$. Thus, if $\beta \in H(Y)$, $\beta \in H(X) \cup H(X \Rightarrow Y)$ and hence $\beta \in H(X \cap (X \Rightarrow Y))$. Therefore, $H(Y) \subseteq H(X \cap (X \Rightarrow Y))$, and by Lemma 7(a), $X \cap (X \Rightarrow Y) \subseteq Y$. Furthermore, $X \Rightarrow Y$ is the greatest such element by definition. By Lemmas 5 and 8, SYST(S*) has a unit and a zero, therefore it is relatively pseudo-complemented.

Corollary. SYST(S*) is a pseudo-Boolean algebra.

Proof. For $X \in SYST(S^*)$, let $\overline{X} = \{s: STR-H(X) \subseteq H(s)\}$. Then $\overline{X} = \{s: H(0)-H(X) \subseteq H(s)\} = (X \Rightarrow 0)$ where 0 is the zero of SYST(S*).

There are a number of ways in which H (i.e., G) can be viewed.

Perhaps the most natural of these is

$$\mathfrak{B} \rightarrow_G s$$

if and only if for r correct in $\mathfrak{B}$,

$$\langle r^\alpha(s,0) : \alpha < \omega \rangle$$

is a terminating process of (S,r). If (p,m) and (p′,m′) each terminate in $\mathfrak{B}$, then so does (p,m);(p′,m′) and

$$H(p,e) = STR,$$

where

$$p(0) = (\varphi,e,1,e,1)$$
$$p(1) = e^*,$$

while

$$H(p',e) = \emptyset,$$

where

$$p'(0) = (\varphi,t,0,t,0)$$

for $t \neq e$ in M_T.

We can in fact show that every G satisfying Axioms (1) and (2) is of the form

$$G = K \wedge \Omega \tag{6}$$

when $\Omega(\mathfrak{B},s)$ if and only if $\langle r^\alpha(s,0) : \alpha < \omega \rangle$ terminates for r correct in $\mathfrak{B}$.

For suppose G does not satisfy (6). Let $H(p,m) \neq \phi$ and $H(p',m') = \phi$. By Axiom (2) we should have

$$H((p,m);(p',m')) = \phi.$$

But since $G \neq K \wedge \Omega$ there is some $\mathfrak{B} \in H(p,m)$ such that

$$\sim\Omega(\mathfrak{B},(p,m)).$$

(In other words, there is always such a choice of p and m.) Thus, if r is correct in $\mathfrak{B}$, there is no $\alpha<\omega$ such that

$$r^{\alpha}(((p,m);(p',m')),0) = (p;p',m'',i)$$

for some $m'' \in M_T$ and $(p;p')(i) = \hat{b}(i)$ where $\hat{b}$ is a bridge program. Then the property G of the process

$$\langle r^{\alpha}(((p,m);(p',m'):\alpha<\omega\rangle$$

in $\mathfrak{B}$ depends only on

$$\langle r^{\alpha}(p,m,0):\alpha<\omega\rangle$$

in $\mathfrak{B}$. Hence

$$H((p,m);(p',m')) \neq \phi.$$

An example of K is obtained by setting

$$\mathfrak{B} \rightarrow_K s$$

if and only if for r correct in $\mathfrak{B}$ we have

$$r^{\alpha}(s,0) = +$$

for all α. It is easy to verify that $K \wedge \Omega$ has the required properties.

Properties Inherited from L

Throughout this section, H will be defined with respect to the termination property. That is, $\beta \in H(s)$ if and only if r is correct in β and

$$<r^{\alpha}(s,0) : \alpha < \omega>$$

is a terminating process of the computer (S,r) in PC.

Let

$$X_0 \subseteq X_1 \subseteq \ldots \subseteq X_n \subseteq \ldots \tag{7}$$

be a chain in 2^{S^*}. The chain is said to be of <u>increasing strength</u> just in case $H(X_i) \not\subseteq H(X_{i+1})$ for all $i \geq 0$. That is, if $X_i \subseteq X_{i+1}$ but there is some $s \in X_{i+1}$ such that it is not the case that

$$X_i \Vdash s.$$

If X is a system in S* and there is no chain of increasing strength, as in (7) for which

$$X = \bigcup_{i \geq 0} X_i,$$

then X is said to be <u>finitely axiomatizable</u>. An immediate result of this definition is that if X is a finitely axiomatizable system, then there is a finite subset Y of X such that Cl(Y) = X. For programs this is equivalent to stating that if a set of programs is closed (by semantic

entailment) with respect to the termination property, then there is one terminating program from which the termination of the rest may be inferred (however complex or nonconstructive the inference may be).

Theorem 10. If L is incomplete, then there is a system X over S* which is not finitely axiomatizable.

Proof. Let $\{\varphi_i : i \in I\}$ be an enumeration of undecidable sentences in L so that for all $j,k \in I$, φ_j and φ_k are not contradictory. For each $i \in I$, let

$$s_{\varphi_i} = (p_{\varphi_i}, e)$$

for

$$p_{\varphi_i}(0) = (\varphi_i, e, 1, t, 0), \quad t \neq e$$

$$p_{\varphi_i}(1) = e^*.$$

To each φ_i, we correspond a system $Y_i = Cl(s_{\varphi_i})$, and notice that

$$\{s: H(s_{\varphi_0}; s_{\varphi_1}) \subseteq H(s)\}$$

$$= \{s: H(s_{\varphi_0}) \cap H(s_{\varphi_1}) \subseteq H(s)\}$$

$$= Cl(Y_0 + Y_1) = Y_0 + Y_1.$$

Hence, we construct a chain of systems based on the Y_i: for all $n \geq 0$ let

$$X_n = \sum_{i=0}^{n} Y_i.$$

By the definition of system union

$$X_i \subseteq X_{i+1}$$

for all $i \geq 0$, so the X_i indeed form a chain of systems in SYST(S*) and hence in 2^{S^*}. By Lemma 6 we have $H(X_{i+1}) \subseteq H(X_i)$. To show the inclusion is proper, consider the program

$$P_{\varphi_0}; P_{\varphi_1}; \ldots; P_{\varphi_n}. \tag{8}$$

For each $\mathfrak{B} \in H(X_i)$ either $\mathfrak{B} \models \varphi_{i+1}$ or $\mathfrak{B} \models \sim\varphi_{i+1}$. Hence no model $\mathfrak{B}$ such that $\mathfrak{B} \models \bigwedge_{j \leq i} \varphi_j \wedge \sim\varphi_{i+1}$ will be in $H(X_{i+1})$, since otherwise (8) would be nonterminating after i iterations. Thus, $H(X_i) \not\subseteq H(X_{i+1})$ and

$$X_0 \subseteq X_1 \subseteq \ldots \subseteq X_n \subseteq \ldots$$

is a chain of increasing strength. Suppose $X = \bigcup_{\alpha \geq 0} X_\alpha$ and that $s \in X$. Then $s \in X_\beta$ for some β and there is a finite subset J of I such that $s \in \sum_{j \in J} Y_j$. Therefore, X is a system in S* and since it is the union of a chain of increasing strength, it is not finitely axiomatizable.

The value of finding a suitable framework for systems has been shown for classical logic. If systems relate to the logical structure of a language in a regular way, then the results concerning them help fix our intuitions about an unknown logic. With the rather simple-minded programming language we have isolated for study, we can show that the control structure -- the

<u>if</u>...<u>then</u>...<u>else</u>...

method of specifying processes -- adds insignificantly to the logical (i.e., algebraic) structure of sentences in L.

Given an oracle (we will later remove this restriction) to decide halting for $(p,m) \in S^*$, we can construct an array of symbols, called a

halting array. For each valuation r for which there is a model in which it is correct and such that

$$<r^{\alpha}(p,m,0):\alpha<\omega> \tag{9}$$

is a terminating process, we enter an r column in the array. If a halting array has no columns, it is called empty. The first entry in the r column is the symbol (ψ_0,m), where if $lp(0) = \varphi$, then

$$\psi_0 = \begin{cases} \varphi_0^+, & \text{if } r(p,m,0) = + \\ \varphi_0^-, & \text{if } r(p,m,0) = - . \end{cases}$$

If (9) terminates after β iterations, then the nth entry ($n<\beta$) in the r column is the symbol (ψ_{n-1},m'), where $r^n(p,m,0) = (p,m',j)$, $lp(j) = \varphi$ and

$$\psi_{n-1} = \begin{cases} \varphi_{n-1}^+, & \text{if } r(p,m',j) = + \\ \varphi_{n-1}^-, & \text{if } r(p,m',j) = - . \end{cases}$$

The βth entry is generated as the nth entry above and is the final entry in the r column. If the r column is identical to a previous column in the array, the r column is deleted from the array.

The proof that the Lindenbaum-Tarski algebra for PL(QF) is a Boolean algebra makes use of the construction of halting arrays, but it involves some rather unpleasant counting. Therefore, we will first give an example which illustrates the method of proof.

We will use $\varphi[x_1,\ldots,x_n]$ to indicate that the variables $x_1,\ldots,x_n$ are all and only those variables which occur free in φ. Similarly, for $t\in T$, $t[x_1,\ldots,x_n]$ exhibits a list of the variables which appear in t, and

likewise for $m[x_1,\ldots,x_n]$ when $m\in M_T$. For a given expression φ the expressions $\varphi[x_1,\ldots,x_n]$ and $\varphi[y_1,\ldots,y_n]$ indicate that the latter differ from φ only in containing the x_i free (the y_i free) for each free occurrence of a variable z_i in φ.

Suppose we are given $(p,m[u])\in S^*$ such that

$$p(0) = (\varphi_1[x],t_1[x,y],1,t_4[x,z],2)$$
$$p(1) = (\varphi_2[y],t_2[z],3,t_3[x],0)$$
$$p(2) = (\varphi_3[u,z],t_5[x],0,t_6[u],3)$$
$$p(3) = e^*$$

and the halting array (with obvious subscripting omitted)

r_0:
$$(\varphi_1^+[x],m[u])$$
$$(\varphi_2^+[y],t_1[x,y]\cdot m[u])$$

r_1:
$$(\varphi_1^-[x],m[u])$$
$$(\varphi_3^-[u,z],t_4[x,z]\cdot m[u])$$

r_2:
$$(\varphi_1^+[x],m[u])$$
$$(\varphi_2^-[y],t_1[x,y]\cdot m[u])$$
$$(\varphi_1^-[x],t_3[x]\cdot t_1[x,y]\cdot m[u])$$
$$(\varphi_3^-[u,z],t_4[x,z]\cdot t_3[x]\cdot t_1[x,y]\cdot m[u])\,.$$

First, we modify the halting array to make the variables which appear in each of the columns distinct from the variables which appear in the other columns. One such modified array is:

r_0: $(\varphi_1^+[x],m[u])$

$(\varphi_2^+[y],t_1[x,y]\cdot m[u])$

r_1: $(\varphi_1^-[x_1],m[u_1])$

$(\varphi_3^-[u_1,z_1],t_4[x_1,z_1]\cdot m[u_1])$

r_2: $(\varphi_1^+[x_2],m[u_2])$

$(\varphi_2^-[y_2],t_1[x_2,y_2]\cdot m[u_2])$

$(\varphi_1^-[x_2],t_3[x_2]\cdot t_1[x_2,y_2]\cdot m[u_2])$

$(\varphi_3^-[u_2,z_2],t_4[x_2,z_2]\cdot t_3[x_2]\cdot t_1[x_2,y_2]\cdot m[u_2])$.

Next, we explicitly give a program $\overline{p}$ which is the complement of p:

$\overline{p}(0) = (\varphi_1[x],t_1[x,y],1,e,2)$

$\overline{p}(1) = (\varphi_2[y],e,9,e,2)$

$\overline{p}(2) = (\varphi_1[x_1],e,4,t_4[x_1,z_1],3)$

$\overline{p}(3) = (\varphi_3[u_1,z_1],e,4,e,9)$

$\overline{p}(4) = (\varphi_1[x_2],t_1[x_2,y_2],5,e,8)$

$\overline{p}(5) = (\varphi_2[y_2],e,8,t_3[x_2],6)$

$\overline{p}(6) = (\varphi_1[x_2],e,8,t_4[x_2,z_2],7)$

$\overline{p}(7) = (\varphi_3[u_2,z_2],e,8,e,9)$

$\overline{p}(8) = e^*$

$\overline{p}(9) = (\psi,t,9,t,9)$, for any $\psi\in L(QF)$, $t \neq e$.

To show that $\overline{p}$ has the properties of a true (Boolean) complement, let $\overline{m} = m[u]\cdot m[u_1]\cdot m[u_2]$. Suppose $\mathfrak{B}\in H(p,m)$ and that r is a valuation correct in $\mathfrak{B}$. Then r generates one of the columns r_0, r_1, r_2. If, for instance, r generates r_1, we have either

$$r(\bar{p},\bar{m},0) = (\bar{p},\bar{m},2)$$

or

$$r^2(\bar{p},\bar{m},0) = (\bar{p},t_1[x,y]\cdot\bar{m},2),$$

since otherwise r would generate the r_0 column. But since r does generate the r_1 column, we have for all $\alpha\geq 0$, either

$$r^{\alpha+3}(\bar{p},t_1[x,y]\cdot\bar{m},2) = (\bar{p},\underbrace{t\cdot\ldots\cdot t}_{\alpha \text{ times}}\cdot t_4[x_1,z_1]\cdot t_1[x,y]\cdot\bar{m},9)$$

or

$$r^{\alpha+2}(\bar{p},\bar{m},2) = (\bar{p},\underbrace{t\cdot\ldots\cdot t}_{\alpha \text{ times}}\cdot t_4[x_1,z_1]\cdot\bar{m},9)$$

so that $\mathfrak{B}\notin H(\bar{p},\bar{m})$. Conversely, suppose $\mathfrak{B}\notin H(p,m)$. Then if r is a valuation correct in $\mathfrak{B}$, it generates none of the array columns. Hence, for some $\alpha\geq 3$,

$$r^{\alpha}(\bar{p},\bar{m},0) = (\bar{p},m'\cdot\bar{m},8),$$

where m′ depends on the choice of r. In other words, $\mathfrak{B}\in H(\bar{p},\bar{m})$. Thus, we have

$$H(\bar{p},\bar{m}) = \text{STR-}H(p,m).$$

In order to prove the fundamental theorem of this section, we will use the following lemma.

Lemma 10. For each $(p,m)\in S^*$, the associated halting array is

either empty or has a finite number of columns.

Proof. For each r, the process

$$<r^{\alpha}(p,m,0):\alpha<\omega>$$

either terminates or not. If the latter holds for every r, then the array is empty. Therefore, suppose that the array associated with $(p,m)\in S^*$ is infinite. Since p is a program there are only finitely many lines p(i) of p such that

$$p(i) = (\varphi,t_1,j_1,t_2,j_2) \neq e^*$$

and either $p(j_1) = e^*$ or $p(j_2) = e^*$, or both. In addition there are only finitely many lines $p(i) \neq e^*$. Thus, by paralleling the argument in Theorem 5, one of the columns in the halting array must be the r column, where for $\alpha<\beta$

$$r^{\alpha}(p,m,0) = (p,t_n'\cdot\ldots\cdot t_1'\cdot m,j),$$

$$r^{\beta}(p,m,0) = (p,t_{k+n}'\cdot\ldots\cdot t_{n+1}'\cdot t_n'\cdot\ldots\cdot t_1'\cdot m,j),$$

where the procedure

$$m_{\alpha} = t_n'\cdot\ldots\cdot t_1'\cdot m$$

exhausts the terms which appear in

$$m_{\beta} = t_{k+n}'\cdot\ldots\cdot t_{n+1}'\cdot t_n'\cdot\ldots\cdot t_1'\cdot m,$$

so that

$$||m_{\alpha}|| = ||m_{\beta}||. \tag{10}$$

Let $\varphi = lp(j)$. Since the r column appears in the halting array, it must be the case that if the αth entry is

$$(\varphi^{+}_{\alpha-1}, m_{\alpha})$$

then the βth entry is

$$(\varphi^{-}_{\beta-1}, m_{\beta}).$$

For if not, then an infinite loop would result and r would not have appeared as a column in the array. Similarly, if the αth entry is

$$(\varphi^{-}_{\alpha-1}, m_{\alpha})$$

then the βth entry is

$$(\varphi^{+}_{\beta-1}, m_{\beta}).$$

But then there is some $\mathfrak{B} \in STR$ such that either

$$\mathfrak{B} \models \varphi[m_{\alpha}, \bar{b}], \text{ for all } \bar{b}$$

and for some $\bar{b}$ not

$$\mathfrak{B} \models \varphi[m_{\beta}, \bar{b}],$$

or

$$\mathfrak{B} \models \varphi[m_{\beta}, \bar{b}], \text{ for all } \bar{b}$$

and for some $\bar{b}$ not

$$\mathfrak{B} \models \varphi[m_{\alpha}, \bar{b}].$$

By equation (10), both of these cases are impossible. Hence there is no such column.

Theorem 11. For each $(p,m) \in S^*$, there is a $(\bar{p},\bar{m}) \in S^*$ such that

$$H(\bar{p},\bar{m}) = \text{STR-}H(p,m).$$

Proof. If the halting array for (p,m) is empty, then $(\bar{p},\bar{m})$ is any $s \in S^*$ such that H(s) = STR, which is known to exist by Axiom (2) for H. Suppose that the halting array for (p,m) is nonempty. Then by Lemma 10, it is finite. Let k be the number of columns in the halting array, and denote the nth entry in the r column by

$$(\psi_n^r, t_n \cdot \ldots \cdot t_1 \cdot m).$$

Assume that the columns are ordered from left to right in order of generation and the r_0 column is the first column, the r_1 column is the second column and so on. We first perform a transformation of free variables in the array as follows. If $j>0$, then entries in the r_j column

$$(\psi_n^{r_j}, t_n \cdot \ldots \cdot t_1 \cdot m)$$

are transformed to

$$(V_{\tau_j}\psi_n^{r_j}, V_{\tau_j}t_n \cdot \ldots \cdot V_{\tau_j}t_1 \cdot V_{\tau_j}m),$$

where no free variable which occurs in the transformed entries occurs free in the r_i column for any $i<j$. Thus, in the new array, no two columns share free variables in either the ψ_n or in the semigroup elements.

The program $\bar{p}$ can now be specified directly from the transformed

halting array. Assume that each r_j column, $j \leq k$, in the halting array contains β_j entries. Then for $i < \beta_0 - 1$,

$$\bar{p}(i) = \begin{cases} (\varphi, t_{i+1}, i+1, e, \beta_0), & \text{if } \psi_i^{r_0} = \varphi_i^+ \\ (\varphi, e, \beta_0, t_{i+1}, i+1), & \text{if } \psi_i^{r_0} = \varphi_i^-, \end{cases}$$

and

$$\bar{p}(\beta_0 - 1) = \begin{cases} (\varphi, e, \sum\limits_{1 \leq k} \beta_1 + 1, e, \beta_0), & \text{if } \psi_{\beta_0}^{r_0} = \varphi_{\beta_0}^+ \\ (\varphi, e, \beta_0, e, \sum\limits_{1 \leq k} \beta_1 + 1), & \text{if } \psi_{\beta_0}^{r_0} = \varphi_{\beta_0}^-. \end{cases}$$

We set

$$\bar{p}(\sum_{1 \leq k} \beta_1) = e^*$$

$$\bar{p}(\sum_{1 \leq k} \beta_1 + 1) = (\varphi, t, \sum_{1 \leq k} \beta_1 + 1, t, \sum_{1 \leq k} \beta_1 + 1),$$

for some $\varphi \in L(QF)$ and $t \neq e$, and for all $i > \sum\limits_{1 \leq k} \beta_1 + 1$

$$\bar{p}(i) = e^*.$$

The rest of $\bar{p}$ is filled in as follows. If $j < k$ and

$$\sum_{1 \leq j} \beta_1 \leq i < \sum_{1 \leq j+1} \beta_1 - 1,$$

then for $\chi_1 = i - \sum\limits_{1 \leq j} \beta_1$,

$$\bar{p}(i) = \begin{cases} (V_{\tau_{j+1}}\varphi, V_{\tau_{j+1}}t_{\chi_i}, i+1, e, \sum\limits_{1 \leq j+1} \beta_1), & \text{if } V_{\tau_{j+1}}\psi^{r_{j+1}} = V_{\tau_{j+1}}\varphi_{\chi_i}^+ \\ (V_{\tau_{j+1}}\varphi, e, \sum\limits_{1 \leq j+1} \beta_1, V_{\tau_{j+1}}t_{\chi_i}, i+1), & \text{if } V_{\tau_{j+1}}\psi_{\chi_i}^{r_{j+1}} = V_{\tau_{j+1}}\varphi_{\chi_i}^-. \end{cases}$$

While if

$$i = \sum_{1 \le j+1} \beta_1 - 1$$

then

$$\bar{p}(i) = \begin{cases} (V_{\tau_{j+1}} \varphi, e, \sum_{1 \le k} \beta_1 + 1, e, i+1), & \text{if } V_{\tau_{j+1}} \psi_{\chi_i}^{r_{j+1}} = V_{\tau_{j+1}} \varphi_{\chi_i}^{+} \\ (V_{\tau_{j+1}} \varphi, e, i+1, e, \sum_{1 \le k} \beta_1 + 1), & \text{if } V_{\tau_{j+1}} \psi_{\chi_i}^{r_{j+1}} = V_{\tau_{j+1}} \varphi_{\chi_i}^{+}. \end{cases}$$

Let $\bar{m} = (m \cdot V_{\tau_1} m \cdot \ldots \cdot V_{\tau_k} m)$. If $\mathfrak{B} \in H(p,m)$ then there is an r correct in $\mathfrak{B}$ which generates the r_j column of the halting array ($j \le k$). Since none of the program segments corresponding to columns in the halting array encounter values produced by previous segments -- free variables having been transformed to prevent this collision -- r will branch to

$$\bar{p}(\sum_{1 \le k} \beta_1 + 1) \tag{11}$$

from line

$$\bar{p}(\sum_{1 \le k} \beta_1 - 1). \tag{12}$$

But the line given in (11) is necessarily a condition for nontermination. Hence, $\mathfrak{B} \notin H(\bar{p},\bar{m})$. On the other hand, if $\mathfrak{B} \notin H(p,m)$ then if r is correct in $\mathfrak{B}$, it generates none of the columns in the halting array. But then for each segment of $\bar{p}$ between

$$\bar{p}(\sum_{1 \le j} \beta_1)$$

and (12), for $j \le k$, r will cause a branch to the segment corresponding to

$$\bar{p}\left(\sum_{1\leq j} \beta_1\right)$$

which is necessarily a condition for termination. Therefore, $\beta \in H(\bar{p},\bar{m})$. It follows that

$$H(\bar{p},\bar{m}) = \text{STR-H}(p,m).$$

We can now turn the construction of halting arrays, and from them arbitrary complements, into an effective procedure. This is done by noticing that if we include in a halting array columns corresponding to different halting paths from those already included in the array, nothing much is changed, since if any valuation generates such a column, that valuation cannot be correct in any model. Obviously adding all such paths is not advisable, so what is required is a constructive means with which to generate a new sort of array.

Theorem 11 gives an effective procedure for passing from any finite array to $\bar{s}$. It is obvious that all such $\bar{s}$ are still Boolean complements. Theorem 12, whose proof we omit, guarantees that such effectively constructible arrays exist for all programs in PL(QF).

Theorem 12. There is an effective procedure which for any $s \in S^*$ yields $\bar{s} \in S^*$, where

$$H(\bar{s}) = \text{STR-H}(s).$$

The algebra of elementary classes of PL(QF), EC, is

$$\{K\text{: for some } s \in S^*,\ H(s) = K\}.$$

We parallel the classical connections between elementary classes and the

logical structure of languages.

Corollary. EC is a Boolean algebra under the usual operations. If L is incomplete, then EC is countable and atomless.

Proof. By Axiom (2), if $H(s) = K_1$ and $H(s') = K_2$, then $H(s';s) = H(s;s') = K_1 \cap K_2$. By Theorem 11, if $H(s) = K$, then $H(\bar{s}) = \text{STR-}K$. Therefore EC is a Boolean algebra. If L is incomplete then EC is countable as a result of Theorem 2 and Lemma 2. Similarly, suppose EC has an atom K. It follows that if $K' \subset K$ is in EC, then $K' = \phi$. But there is some $s \in S^*$ such that $H(s) = K$. Let φ be a sentence of L such that K is not the (classical) elementary class corresponding to φ. We then modify p, if $s = (p,m)$, by replacing each exit line p(j) of p by

$$(\varphi, e, k, t, j)$$

where $p(k) = e^*$. Then if p' is this new program, we have $H(p',m) \subset H(s)$, but not necessarily that $H(p',m) = \phi$.

Corollary. Let F(EC) be the collection of all filters in EC. There are lattice operations on F(EC) such that F(EC) is (lattice) isomorphic to SYST(S*).

Proof. Define a mapping d as follows: for all $s \in S^*$,

$$d(s) = \{s' : H(s) = H(s')\}.$$

Clearly $\text{SYST}(S^*) \cong \text{SYST}(\{d(s) : s \in S^*\})$ when

$$d(s \cap s') = d(s) \cap d(s')$$

$$d(s + s') = d(s) + d(s').$$

For $\nabla_1, \nabla_2 \in F(EC)$ we have $\nabla_1 \cap \nabla_2 \in F(EC)$ and if

$$\nabla_3 = \cap\{\nabla\colon \nabla\in F(EC) \text{ and } \nabla_1\cup\nabla_2\subseteq\nabla\}$$

then

$$\nabla_1 + \nabla_2 = \nabla_3\in F(EC).$$

Let

$$d^*\colon SYST(\{d(s)\colon s\in S^*\}) \cong EC$$

so that $d^*(d(s)) = H(s)$. The correspondence d^* is extended to an isomorphism in the obvious way.

It is sometimes easier to work with $S^*/\equiv(H)$, where

$$s \equiv s'(H) \text{ if and only if } H(s) = H(s')$$

than with EC itself. Since $S^*/\equiv(H) \cong EC$, there is nothing lost in this translation. In fact, $S^*/\equiv(H)$ is the Lindenbaum-Tarski algebra $LTA(S^*)$. By the first corollary to Theorem 12 we have $LTA(S^*) \cong LTA(L)$. Pursuing further this analogy between L and S* we now show that Ω is a complete semantic concept. That is, the following theorem demonstrates that in order to fully understand the semantic structure of the programming language PL(QF) it is necessary and sufficient to examine the terminating processes which the programs of PL(QF) specify.

Theorem 13. For each computer $(S,r)\in PC$ there exists an ultrafilter ∇ in EC such that for all $s\in S$,

$$r(s) = + \text{ iff } \{\beta\colon \text{for } r' \text{ correct in } \beta,\ r'(s) = +\}\in\nabla. \qquad (13)$$

Conversely, for each ultrafilter ∇ in EC there is an abstract computer

$(S,r)\in PC$ such that (13) holds for all $s\in S$.

Proof. Suppose $(S,r)\in PC$. Then let

$$\nabla = \{K: K\in EC \text{ and } \mathfrak{B}\in K\},$$

which is clearly an ultrafilter in EC. We will construct a model $\mathfrak{B}'$ so that the valuation given in (13) is correct in $\mathfrak{B}'$, for each choice of ultrafilter ∇ in EC. Let $\mathfrak{B}' = \langle B', F, R\rangle$ where B' is the set of terms of L(QF). For each function symbol f_γ of L(QF), let

$$\underline{f}_\gamma(t_1,\ldots,t_{\mu_1(\gamma)}) = f_\gamma(t_1,\ldots,t_{\mu_1(\gamma)}),$$

and let each individual constant denote itself. Finally, we impose the following restriction on relations in R: for each $m\in M_T$

$$\langle \|m\|\bar{b}(t_1),\ldots,\|m\|\bar{b}(t_{\mu_2(\gamma)})\rangle \in q_{\underline{\gamma}}, \text{ for all } \bar{b}$$

if and only if

$$\{\mathfrak{B}:\mathfrak{B} \models q_\gamma(t_1,\ldots,t_{\mu_2(\gamma)})[m,\bar{b}], \text{ for all } \bar{b}\}\in\nabla.$$

The model $\mathfrak{B}'$ can be "contracted" to a normal model $\mathfrak{B}$ (see, e.g., [36], p. 80 and also [1], p. 105) such that $\mathfrak{B}$ is in STR. This is so, since if φ is valid, then

$$\{\mathfrak{B}:\mathfrak{B} \models \varphi[m_\varphi,\bar{b}], \text{ for all } \bar{b}\in B^T\} = STR$$

and hence

$$\mathfrak{B} \models \varphi[m_\varphi,\bar{b}], \text{ for all } \bar{b}.$$

By Lemma 4 there is a valuation correct in $\mathfrak{B}$. A simple induction shows

that (13) holds for all $s \in S$.

S* is compact with respect to EC just in case for any subset X of S*, the elements of X are simultaneously successful in some model if and only if every finite subset of X succeeds in some model. That S* is, in fact, compact for EC is not surprising, and we omit the proof since it follows trivially from the compactness of L and Axiom (2) for H. It is worthwhile to note, however, that a compactness theorem for PL(QF) is a metalogical result about PL(QF) which does not require reference to the syntactic structure of PL(QF) (e.g., no reference to deduction is involved). These sorts of results given in sufficient numbers and specific enough form are the most obvious candidates for fixing the logical syntax of PL(QF).

Theorem 14. S* is compact for EC. In particular, for every $X \subseteq S^*$, $H(X) \neq \emptyset$ just in case for all $Y \subseteq X$, $Card(Y) < \omega$, $H(Y) \neq \emptyset$.

The statement of Theorem 14 can be strengthened slightly by noting that if $H(Y) \neq \emptyset$ for all finite $Y \subseteq X$, then there is an ultraproduct [1] $\mathfrak{B}$ over $\bigcup_{Y \subseteq X} H(Y)$ such that $\mathfrak{B} \in H(X)$.

Complex Programs

As we have remarked several times in preceding sections PL(QF) contains few "useful" programming features. There are three sorts of complexities which may be introduced to PL(QF) to make it more useful. These are additions to the "compile time" characteristics of PL(QF), to the "execution time" characteristics of PL(QF) and to the way in which the semantic theory itself treats a program. In this section we will make some obvious extensions of PL(QF) in the first category. Indeed,

we will find that the Boolean logic of the previous section is applicable to more complicated programming structures in a natural way.

The most significant feature of the notion of procedure has been that for any procedure t,

$$\|t\|^n = \|t\|, \quad n \geq 1.$$

Suppose now that we modify PL(QF) to contain a procedure

$$x \leftarrow t$$

(called the assignment of t to x) whenever x is an individual variable and t is a procedure. We specify the following intended interpretation: if t is a term, then

$$\|x \leftarrow t\|\bar{a}(t_0) = \begin{cases} \bar{a}(t_0), & \text{if } t_0 \neq x \\ \|t\|\bar{a}(t), & \text{if } t_0 = x. \end{cases}$$

If we have $t = f(x)$, and

$$\vdash_L \sim(\forall x)(x = f(x)),$$

then

$$\|x \leftarrow t\|^n \neq \|x \leftarrow t\|, \quad n \geq 2.$$

Actual procedure calls then take on a new significance; for example, in

0: if $\varphi(x)$ then [$y \leftarrow f(z)$; go to 1] else [$y \leftarrow g(x)$; go to 2];

1: if $\varphi(y)$ then [do $h(y,z)$; go to 2] else [do $h(y,x)$; go to 4];

2: if $\psi(x,y,z)$ then [$y \leftarrow z$; go to 1] else [$z \leftarrow x$; go to 0]; ,

since in an application to a particular state, we have calls on

$$h(f(z),z), h(g(x),x), h(z,z), h(f(z),x), \ldots$$

the approach of the previous section is no longer obviously applicable.

We can make some notational changes. For each program p (with or without assignments), let n_p be the number of distinct terms which appear in p and let $\bar{\xi}_p$ be the ordered n_p-tuple of distinct terms appearing in p (as before, ordered first by length and then by lexicographic means). We isolate a subset S' of S^* such that $(p,m) \in S'$ just in case

$$m = n_p \bar{\xi}_p \cdot \ldots \cdot 1 \bar{\xi}_p .$$

Therefore, we may denote elements $(p,m) \in S'$ by p with no danger of confusion. A property G of a process is now a property of

$$\langle r^{\alpha}(p,m,0) : \alpha < \omega \rangle$$

where $(p,m) \in S'$, so that it makes sense to declare

$$\mathfrak{B} \rightarrow p \text{ or } \mathfrak{B} \not\downarrow H(p)$$

when p is a program (see also Theorem 1).

By a program we now mean a program with assignment. Denote by $y_1, \ldots, y_n$ those individual variables which do not appear on the left hand sides of assignments of the form $y \leftarrow t$ in p and by $x_1, \ldots, x_n$ those

individual variables which do appear on the left-hand sides of assignments. In reality the x_i and y_i need not always be distinct, but by transforming variable names we can always satisfy this criterion.

Our first result for programs with assignment is achieved by applying a method due to Manna [26]. Let $\mathfrak{B} \in STR$ and let r be correct in $\mathfrak{B}$. We provide an extension of L(QF) to a language L(MV) and a corresponding extension of each $\mathfrak{B} \in STR$ as follows. Let p be a program and $i \in K_p$. The relation

$$\delta_{p(i)} \subseteq B^{n_p} \tag{14}$$

is said to be a <u>valid $\mathfrak{B}$ predicate for p(i)</u>† just in case there is some $\alpha < \omega$ such that for $(p,m) \in S'$

$$r^{\alpha}(p,m,0) = (p,m',i) \tag{15}$$

only if for all $\bar{b} \in B^T$

$$\langle ||m'||\bar{b}(1\bar{\xi}_p), \ldots, ||m'||\bar{b}(n_p\bar{\xi}_p)\rangle \in \delta_{p(i)}. \tag{16}$$

The relation (14) is called a <u>minimal</u> valid $\mathfrak{B}$ predicate for p(i) just in case (15) holds for some α if and only if for all $\bar{b} \in B^T$, (16) holds. Notice that if $\mathfrak{B} \nmid H(p)$ and if i is an exit for p, then $\delta_{p(i)} = \emptyset$. In other words, if $\mathfrak{B} \nmid H(p)$ and if q is a predicate symbol of L(MV) for which $\underline{q} = \delta_{p(i)}$,

---

†This is Manna's device and terminology [26]. There should be no confusion over the use of the word "predicate" to describe an actual relation in $\mathfrak{B}$. We still reserve "predicate symbol" for the corresponding symbol in L(MV).

$$\mathfrak{B} \models (\forall z_1)(\forall z_2)\ldots(\forall z_{n_p})\sim q(z_1,\ldots,z_{n_p}).$$

Let K_p support p. For each $i \in K_p$, let $q_{p(i)}$ be an n_p-ary predicate symbol. For each line p(i), $i \in K_p$, define two formulas of the extended language L(MV) which will eventually play roles analogous to the halting array columns for languages with assignment:

$$W^+_{p(i)} \equiv (q_{p(i)}(\bar{\xi}_p) \wedge 1p(i)) \supset q_{p(3p(i))}(\bar{\xi}_p/2p(i))$$

$$W^-_{p(i)} \equiv (q_{p(i)}(\bar{\xi}_p) \wedge \sim 1p(i)) \supset q_{p(5p(i))}(\bar{\xi}_p/4p(i)),$$

where for $1 \leq j \leq n_p$,

$$j(\bar{\xi}_p/x \leftarrow t) = \begin{cases} j\bar{\xi}_p, & \text{if } j\bar{\xi}_p \neq x \\ t, & \text{if } j\bar{\xi}_p = x, \end{cases}$$

and for some sentence φ

$$q_{p(i)}(\bar{\xi}_p) = \begin{cases} \varphi \vee \sim\varphi, & \text{if } i = 0 \\ \varphi \wedge \sim\varphi, & \text{if } i \text{ is an exit of } p. \end{cases}$$

Then for

$$K = \{i: i \in K_p \text{ and } p(i) \neq e^*\},$$

define

$$W_p \equiv (\forall x_1)\ldots(\forall x_n)(\bigwedge_{i \in K} W^+_{p(i)} \wedge \bigwedge_{i \in K} W^-_{p(i)}),$$

where the x_i, $1 \leq i \leq n$, are as defined above.

Notice that those individual variables which are "bound" in the

program p by appearing on the left hand side of an assignment statement are accordingly the bound variables of the formula W_p.

Theorem 15. $\mathfrak{B} \to p$ if and only if $\mathfrak{B} \models \sim W_p$.

Proof. Let r be correct in $\mathfrak{B}$ and for $(p,m) \in S'$, let

$$\pi = \langle r^{\alpha}(p,m,0) : \alpha < \omega \rangle.$$

Suppose π is a nonterminating process of (S,r). Let $\underline{q}_{p(i)} = \delta_{p(i)}$ in $\mathfrak{B}$ where $\delta_{p(i)}$ is a minimal valid predicate for p(i). Set

$$\overline{b} = ||m||\overline{b'}$$

for some $\overline{b'} \in B^T$ and let $\langle b \rangle \in B^{\omega}$ be such that for each variable z,

$$\overline{b}(z) = \langle b \rangle (v(z)),$$

where v is the natural correspondence between variables and ω. If not

$$\mathfrak{B} \models {}_{\langle b \rangle} W_p$$

then for some $i \in K$, it is not the case that

$$\mathfrak{B} \models {}_{\langle b \rangle} W^{+}_{p(i)} \wedge W^{-}_{p(i)}.$$

If there is no α such that $3\pi(\alpha) = i$, then since $\delta_{p(i)}$ is valid in $\mathfrak{B}$

$$q_{p(i)}(\overline{\xi}_p) = \varphi \wedge \sim\varphi,$$

and hence not

$$\mathfrak{B} \models {}_{\langle b \rangle} ((\varphi \wedge \sim\varphi) \wedge 1p(i)) \supset q_{p(3p(i))}(\overline{\xi}_p / 2p(i)) \wedge ((\varphi \wedge \sim\varphi) \wedge \sim 1p(i)) \supset q_{p(5p(i))}(\overline{\xi}_p / 4p(i)) \tag{17}$$

which is impossible. On the other hand, if $i = 3\pi(\alpha)$ for some α, then

$$\mathfrak{B} \models_{\langle b\rangle} q_{p(i)}(\bar{\xi}_p).$$

If $\mathfrak{B} \models_{\langle b\rangle} 1p(i)$, then

$$\mathfrak{B} \models_{\langle b\rangle} q_{p(3p(i))}(\bar{\xi}_p/2p(i)),$$

while if $\mathfrak{B} \models_{\langle b\rangle} \sim 1p(i)$, then

$$\mathfrak{B} \models_{\langle b\rangle} q_{p(5p(i))}(\bar{\xi}_p/4p(i)).$$

A contradiction also arises from this. Thus, in either case we have a contradiction and we conclude that W_p is satisfiable in $\mathfrak{B}$.

Conversely, suppose

$$\mathfrak{B} \models_{\langle b\rangle} W_p$$

for some $\langle b\rangle \in B^{\omega}$. By the construction of W_p, $q_{p(i)}$ is a valid $\mathfrak{B}$ predicate for $p(i)$. Therefore

$$q_{p(i)}(\bar{\xi}_p) \equiv \varphi \wedge \sim\varphi$$

if i is an exit of p. If $\mathfrak{B} \rightarrow p$ then for some $j \in K_p$ either $3p(j) = i$ or $5p(j) = i$ and $3\pi(\alpha) = j$ for some $\alpha < \omega$. But then either

$$\mathfrak{B} \models_{\langle b\rangle} q_{p(i)}(\bar{\xi}_p) \wedge 1p(i)$$

or

$$\mathfrak{B} \models_{\langle b\rangle} q_{p(i)}(\bar{\xi}_p) \wedge \sim 1p(i).$$

In either case we have a contradiction since

$$\mathfrak{B} \models_{\langle b \rangle \sim q_{p(i)}} (\bar{\xi}_p).$$

Therefore $\mathfrak{B} \notin H(p)$.

Theorem 16. If L(QF) = L(MV), then for each program p with assignments there is a program $\bar{p}$ such that

$$H(\bar{p}) = \text{STR-H}(p).$$

Proof. This follows trivially from Manna's theorem and Theorem 11, since for

$$p_W(0) = (W_p, t, 0, e, 1)$$
$$p_W(1) = e^*$$

where $e \neq t \in T$, we have

$$\mathfrak{B} \to p \text{ if and only if } \mathfrak{B} \to p_W$$

and there is a program $\bar{p}_W$ so that

$$H(\bar{p}_W) = \text{STR-H}(p_W).$$

Corollary. If L(QF) = L(MV), then the Lindenbaum-Tarski algebra LTA(S′) for programs with assignment is a Boolean algebra. If L is incomplete, this algebra is countable and atomless.

Proof. This is essentially the result of the previous section. The existence of complements is guaranteed by Theorem 16, and as before we have

$$H(p;p') = H(p) \cap H(p').$$

The hypothesis of Theorem 16 is generally uninvestigated. Should it fail, there is no assurance that LTA(S′) is even a lattice. Since we have given meets as primitive and have been assured that joins could always be obtained from meets and complements, there is no obvious reason to believe that LTA(S′) is <u>always</u> a lattice. The next theorem demonstrates that this is, in fact, the case for all languages with assignment.

<u>Theorem 17</u>. LTA(S′) is a lattice with greatest and least elements.

<u>Proof</u>. The existence of meets and greatest and least elements follows directly from previous results. We now give a construction which accounts for the join of two elements in S′. For simplicity, we deal only with programs. It will then be obvious how to complete the proof.

Let p and p′ be given programs. We may suppose that p and p′ share no variables and that K_p and $K_{p'}$ are initial segments of I. Let $P = \mathrm{Card}(K_p)$ and $P' = \mathrm{Card}(K_{p'})$. Since L(QF) is QF we may also assume the existence in L(QF) of denumerably many individual constants $a_0, a_1, \ldots, a_n, \ldots, b_0, \ldots$, together with the following pair of theorems for each i and j:

$$\vdash_L \sim a_i = a_j \qquad\qquad \vdash_L \sim b_i = b_j$$

when $i \neq j$.

We now explicitly define a program p_v for each line $p_v(j)$. Let x_p and $x_{p'}$ be variables which do not occur in either p or p′:

(1) If $j<P$, then

$$p_v(j) = \begin{cases} e^*, \text{ if } p(j) = e^* \\ (\varphi, t_1, P+i_1, t_2, P+i_2), \text{ if } p(j) = (\varphi, t_1, i_1, t_2, i_2). \end{cases}$$

(2) If $j = P + m$ $(m = 0,\ldots,P-1)$, then

$$p_v(j) = (\varphi, x_p \leftarrow a_m, 2P, x_p \leftarrow a_m, 2P),$$

for any $\varphi \in L(QF)$.

(3) If $j = 2P + m$ $(m = 0,\ldots,P'-1)$, then

$$p_v(j) = (x_{p'} = b_m, e, 2P + P' + m, e, j + 1).$$

(4) If $j = 2P + P' + m$ $(m = 0,\ldots,P'-1)$, then

$$p_v(j) = \begin{cases} e^*, \text{ if } p'(m) = e^* \\ (\varphi, t_1, 2P' + i_1, t_2, 2P' + 2P' + i_2), \text{ if } p'(m) = (\varphi, t_1, i_1, t_2, i_2). \end{cases}$$

(5) If $j = 2P + 2P' + m$ $(m = 0,\ldots,P'-1)$, then

$$p_v(j) = (\varphi, x_{p'} \leftarrow b_m, 2P + 3P', x_{p'} \leftarrow b_m, 2P + 3P'),$$

for any $\varphi \in L(QF)$.

(6) If $j = 2P + 3P' + m$ $(m = 0,\ldots,P-1)$, then

$$p_v(j) = (x_p = a_m, e, m, e, j + 1).$$

(7) If $j \geq 3P + 3P'$, then $p_v(j) = e^*$.

Now, let INIT be a program such that

$$\text{INIT}(0) = (\varphi, x_p \leftarrow a_0, 1, x_p \leftarrow a_0, 1)$$

$$\text{INIT}(1) = (\varphi, x_{p'} \leftarrow b_0, 2, x_{p'} \leftarrow b_0, 2)$$

$$\text{INIT}(2) = e^*,$$

and define the relabeling σ such that for all $i \in I$, $\sigma i = i + 2$. Then the

join of p and p′ is the program p″:

$$p''(j) = \begin{cases} \text{INIT}(j), & \text{if } j = 0,1 \\ L_{\sigma} p_{v}(j), & \text{if } j \geq 2. \end{cases}$$

By the construction just given, p″ succeeds if and only if at least one of p and p′ succeeds.

Satisfaction

We have just seen how addition to the compile time complexity of PL(QF) can be incorporated into the semantics of PL(QF). Another sort of complexity occurs in an expansion of the semantic theory surrounding PL(QF). The motivation for considering this second sort of complexity follows from the observation that if r is correct in $\mathfrak{B}$ and π is a process of (S,r) with $\pi(0) = (p,m,0)$, then there is exactly one path through p which is achievable in $\mathfrak{B}$ with an initial data transformation m. We will assume that PL(QF) contains assignment.

Let y_i, $1 \leq i \leq n$, denote an individual variable bound by assignment as described in the previous section. If m is of the form

$$y_1 \leftarrow t_1 \cdot y_2 \leftarrow t_2 \cdot \ldots \cdot y_n \leftarrow t_n, \tag{18}$$

then m is the intuitive "input" which p requires for execution and a single path <u>should</u> be described. When m is not given by (18), the single-path condition arises by default only.

We redefine the notion of valuation accordingly. Let r be a valuation. r is said to be <u>correct in $\mathfrak{B}$ at $\bar{b} \in B^T$</u>, and is denoted $r_{\bar{b}}$, just in case for all $(p,m,i) \in S$,

$$r_{\overline{b}}(p,m,i) = +$$

if and only if

$$\mathfrak{B} \models 1p(i)[m,\overline{b}].$$

Thus, for each $\overline{b} \in B^T$ we have

$$\pi_{\overline{b}}(\alpha) = \begin{cases} (p,m,0), & \text{if } \alpha = 0 \\ r_{\|2\pi(\alpha-1)\|\overline{b}}((\pi_{\overline{b}}(\alpha-1))), & \text{otherwise.} \end{cases}$$

Similarly, for each choice of G satisfying Axioms (1) and (2),

$$\mathfrak{B} \rightarrow_G p[\overline{b}], \tag{19}$$

just in case for $1\pi_{\overline{b}}(\alpha) = p$ in S', $\pi_{\overline{b}}$ has the property G.

An immediate consequence of the definitions is that (19) holds if and only if

$$\mathfrak{B} \rightarrow_G L_\sigma p[\overline{b'}],$$

where $L_\sigma p$ is the program obtained from p by a relabeling σ such that if $p(0) = (\varphi, t_1, j_1, t_2, j_2)$, then

$$\sigma^{-1}0 = \begin{cases} j_1, & \text{if } \mathfrak{B} \models \varphi[m,\overline{b}] \\ j_2, & \text{if } \mathfrak{B} \models \sim\varphi[m,\overline{b}] \end{cases}$$

and

$$\overline{b'} = \begin{cases} \|t_1\|\overline{b}, & \text{if } \mathfrak{B} \models \varphi[m,\overline{b}] \\ \|t_2\|\overline{b}, & \text{if } \mathfrak{B} \models \sim\varphi[m,\overline{b}]. \end{cases}$$

In Chapter II, the obvious procedure was to associate with each line p(i) of a program p the values + and - independent of the state of the universe input to p. We then showed how this two-valued association gave rise to a natural Boolean-based logic. We noted above that the assignment of + and - values to lines of a program can be made to depend on states of a universe. Furthermore, in doing this, we find that much of the redundance in the description of program execution in Chapter II can be eliminated.

Notice that if

$$\pi = \langle r^{\alpha}(p,m,0) : \alpha < \omega \rangle$$

and if r is correct in $\mathfrak{B}$ and if $\mathfrak{B} \rightarrow p$, then it does not necessarily follow that if $\mathfrak{B} \rightarrow p[\bar{b}]$ for all states $\bar{b}$ that $\pi_{\bar{b}} = \pi$ for all $\bar{b}$. Clearly, we can now adopt a more realistic attitude for choosing "sentences" of PL(QF). For each $\mathfrak{B} \in STR$, let $PR_p(\mathfrak{B}) = \{\pi_{\bar{b}} : \bar{b} \in B^T$ and $\pi_{\bar{b}}(0) = (p,m,0)\}$. Then a program $p \in PL(QF)$ is a <u>sentence</u> (i.e., an <u>imperative</u> sentence) of PL(QF) if and only if for all $\mathfrak{B} \in STR$, $Card(PR_p(\mathfrak{B})) = 1$. Let SC be the set of all sentences of PL(QF). We have some immediate examples of members of SC:

(1) Let $p(i) = (\varphi, y_{i+1} \leftarrow a_i, i+1, y_{i+1} \leftarrow a_i, i+1)$, for any $\varphi \in L(QF)$, individual constant a_i, and $0 \leq i \leq n-1$.

(2) Let $p(i) = e^*$, for all i.

(3) For each $i \in K_p$, let $1p(i)$ be a sentence of L(QF).

Obviously, if we restrict the results of this chapter to SC, the algebra of elementary classes EC remains a Boolean algebra (or a lattice) of the desired sort.

CHAPTER IV

CONCLUDING REMARKS

Execution Time Complexity

In contrast to the "static" nature of program execution described in Chapters II and III are the problems associated with execution time interpretations of certain aspects of program syntax. Examples of these include the class of problems which arise from block structuring (with declarations and scope identifiers, principally) and the class of problems which arise because of procedure declarations (including those associated with parameter call-by-name and call-by-value). We will confine ourselves to problems in the former class.

The motivation for the semicolon operator can be paraphrased: if we interpret p_1 and p_2 as blocks with no global variables, then $p_1;p_2$ is to be interpreted as the block p_3,

```
begin
   begin p1 end;
   begin p2 end
end,
```

so that, independent of the order of execution of p_1 and p_2, the termination of p_3 depends only on the joint termination of p_1 and p_2.

Unfortunately, the construction of $p_1;p_2$ which we described in Chapter III leaves us lacking in intuition when we try to discuss

semantic theories with axioms of the following sort:

$$\mathfrak{B} \rightarrow p_1;p_2[\bar{b}] \text{ if and only if } \mathfrak{B} \rightarrow p_1[\bar{b}] \text{ and } \mathfrak{B} \rightarrow p_2[\bar{b}]. \tag{20}$$

An anomaly arises since it is not always the case that $||V_\tau m||^{\bar{b}}(\tau x) = ||m||^{\bar{b}}(x)$ and, hence, not

$$\mathfrak{B} \models_{<b>} \varphi(x) \equiv \varphi(\tau x).$$

This problem did not arise earlier since choices of +,- branches were not made state-by-state. In other words, while simply transforming variables names may be sufficient to describe programs in SC when they are conjoined as blocks, it is certainly not sufficient for "state-sensitive" treatments of programs.

One way around this difficulty is implicit in Johnston's contour model of block structured processes [18]. Consider, instead of individual valuations $r_{\bar{b}}$ for each $\bar{b} \in B^T$, the "processes" generated by the correspondence

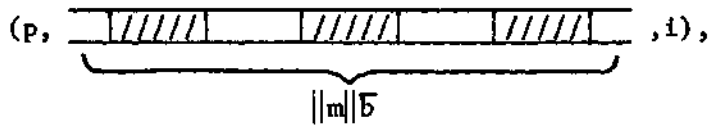
$$r_{\bar{b}}(p,m,0) \Leftrightarrow r(p,||m||^{\bar{b}},0),$$

$$r^2_{\bar{b}}(p,t^{\pm}\cdot m,i^{\pm}) \Leftrightarrow r^2(p,||t^{\pm}||\cdot||m||^{\bar{b}},i^{\pm})$$

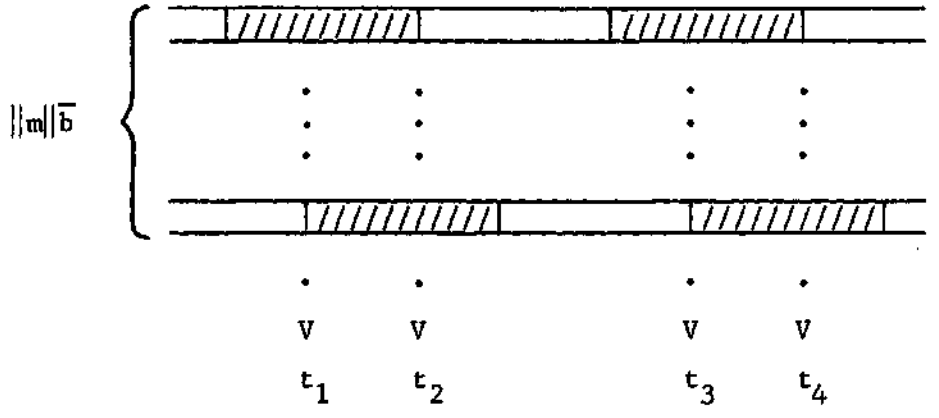
$$\vdots$$

Then, in Johnston's sense, for each $\alpha < \omega$ $\pi_{\bar{b}}(\alpha)$ is a "snapshot" of a computational process containing ". . . a time-invariant algorithm and a time varying record of execution. . . ."

In general $\pi(\alpha)$ might be viewed as the triple



where the hatched areas represent the workspace required by p. Then for programs p_1 and p_2 with overlapping workspaces,



we devise an <u>allocator</u> program $AL(p_1,p_2)$ which for specified $V_\tau p_2$ (resp. $V_\tau p_1$) <u>stores</u> initial values from the intervals (t_1,t_2) and (t_3,t_4). Then $p_1;p_2$ can be viewed in block structured form as:

<u>begin</u>

$AL(p_1,p_2)$;

<u>begin</u> p_1 <u>end</u>;

<u>begin</u> p_2 <u>end</u>

<u>end</u>.

Clearly, by formalizing this procedure, we have condition (20).

<u>Summary</u>

We began with three assumptions:

(1) the intended interpretation of a program should be a process;

(2) processes are experiments carried out in a universe specified by a "base language" L;

(3) it makes sense to ascribe certain properties to programs and models of L based on the results of these experiments in the models.

The bulk of Chapter II was concerned with showing how to directly apply the definitions and methods of model theory to obtain a reasonable sense of "meaning" for programs and with motivating certain departures from standard methods. Theorem 5 provided the impetus for later results concerning finitely recoverable processes for languages without assignment by demonstrating that, from an extensional point of view, all such processes become periodic. This result was applied in Theorem 6 and appears in another form in Chapter III.

In Chapter III, we developed two logical connectives for programs, corresponding to the conjunction and negation of programs. The conjunction operator was introduced axiomatically and corresponded in a natural way to block structuring. Negation was proved to exist in Theorem 11 for programs without assignment and, in a weaker sense, in Theorem 16 for programs with assignment. Disjunction was considered to be separately definable for programs with assignments in Theorem 17.

The fundamental result of Chapter III is implicit in Theorems 11, 16 and 17: it is possible to construe the logical syntax of a programming language to be Boolean-based (bivalent, in the terminology of [49]). Indeed, there is a natural correspondence between the way in which we assign + and - values to decisions in the program and the assignment of true values to the formulas about which the decisions are to be made. It

is, therefore, not surprising that semantic properties of the bivalent base language are passed on to the programming language. Some of these properties were discussed in Chapter III.

Furthermore, by relaxing the restrictions on interpreting all programs as sentence-like forms, we obtained a notion of satisfaction which replaced the more absolute notion of success.

Future Research

In what must be counted as the first rigorous discussion of programming languages for digital computers, J. von Neumann and H. Goldstein (1948, "Planning and Coding Problems for an Electronic Computing Instrument," Report prepared for U.S. Army Ord. Dept.) gave particular emphasis to the logical nature of programming languages:

> Since coding is not a static process of translation, but rather the technique of providing a dynamic background to control the automatic evolution of a meaning, it has to be viewed as a logical problem and one that represents a new branch of formal logics.

In this section, we will discuss some areas of research suggested by this perception in light of developments obtained in this dissertation. Continuing the arguments based on the value of a logical syntax for programming languages, a principal aim of investigations in the semantics of programming languages may be viewed as the recovery of enough logical structure to permit properties of classes of programs to be proved inductively from properties of component programs. In relation to classical logics the activity we have in mind takes the following form. The usual approach in logic is to take an uninterpreted system of logical constants and expressions and to expand on the notion of

interpretation from this basis. On the other hand, we take the notion of interpretation as primitive and, given a suitable description of expressions (i.e., programs), obtain logical constants which are in some sense adequate in the interpretations.

We paraphrase the principal results of Chapter III. Let us identify the logical structure of a programming language with its algebraic structure (i.e., the algebraic structure of its algebra of elementary classes). Then the logical structure of a programming language is never less than that of a lattice, and often is that of a Boolean algebra. For extremely simple programming languages, this result has an even stronger statement: the logical structure is exactly that of classical propositional logic.

There is an interesting observation to be made here. By Theorem 17, every programming language is inherently equipped with a logical "and" and a logical "or" with the usual intended interpretations modified so that it makes sense to command "do p and do p′" and "do p or do p′." In a natural way the conjunction of two programs corresponds to a command that the programs be executed in parallel (or, at least, in a way which closely resembles parallelism). Similarly, the disjunction of two programs results in a nondeterministic command. What is to be noticed about this result is that parallelism and nondeterminism are logically dual notions. This is not at all expected, since we intuitively pair the notion of parallel execution with the notion of serial execution and the notion of nondeterminism with the notion of (strict) deterministic outcome. Apparently, the logical structure when developed along the lines of truth functional logic has little to say about processes that we

regard as serial, inductive or recursive.

We suggest that a solution to this problem rests on an analysis of how recursion relates to the developed logical properties of programs. In other words, it may be possible to impart additional structure to EC so that the essential combinations of blocks with global variables (for example, nesting of blocks) are recoverable within the formal framework. An analogous problem existed in modal logic. The medieval "modes" were discovered not to be truth functional in character (that is, none of the following were to be valid: $\Box p \equiv \sim p$, $\Box p \equiv p$, $\Box p \equiv (p \vee \sim p)$, $\Box p \equiv (p \wedge \sim p)$). Therefore, a set of non-truth functional connectives was introduced to make up an intuitively -- and, as has been recently demonstrated, formally -- acceptable calculus for modal logics by extending classical propositional logic to include this set. It is entirely possible that the activities of "passing along" values from one program segment to the next may relate to truth functional structures (we are here equating "truth" with "success" and "falsity" with "failure") in only a most artificial way. Our previous results notwithstanding, there is no reason to believe that the ad hoc imposition of an algebraic structure [2,8] would result in a less comfortable theory. In either case, the analysis of how program segments combine to preserve a less rigid structure in EC would be in order. These sorts of investigations are central to a number of related questions in the recovery of computational schemes (e.g., the structure of degrees of unsolvability, the structure of complexity classes of algorithms, Scott's theory of lattice theoretic approximation), and we anticipate considerable activity in this area.

In quite a different direction, the semantic properties of

programming languages with assignment are almost totally uninvestigated, and only a handful of results are known. There are many examples of questions which, upon first inspection, should be decided either positively or negatively. For instance, Engeler [10] has noted a certain infinitary quality of programs with assignment. It is unknown, at the present time, whether or not the compactness result of Theorem 7 holds for programming languages with assignment. If not, it would be interesting to ask whether compactness fails for programming languages in the same way which it fails in $L\omega_1\omega$ (see [1], p. 289 ff.).

The basis for deciding that a property or class of properties be meaningfully ascribed to a program is a problem whose solution is presupposed in the discussion of logical syntax. We have already suggested a plausible answer. Let G be a property of an experiment which may be carried out in a number of universes. If an experiment π has the property G in $\mathfrak{B}$ declare $G(\mathfrak{B},\pi)$ and if, furthermore, π is related to a program p in an essential way declare:

$$\mathfrak{B} \in H(p) \text{ if and only if } G(\mathfrak{B},\pi).$$

Thus, we have at hand a method of the intuitively required kind: $\mathfrak{B} \rightarrow p$ if and only if $\mathfrak{B} \in H(p)$ if and only if $G(\mathfrak{B},\pi)$. While this approach is demonstrably (see, e.g., Theorem 13) workable, it is not difficult to devise arguments which indicate that the details of our approach exhibit a certain narrowness in scope. For example, G, as shown in Chapter III, is irrevocably tied to termination, which in itself may not always be an interesting property of processes. Furthermore, if we look at processes in Johnston's extended sense (cf. "Execution Time Complexities" in this

chapter), then we bind the behavior of an experiment to the manner in which we express it. Thus we have introduced a syntactic bias into what was to be a purely semantic system. These arguments give rise to more open questions concerning the semantics of PL. For example, let an experiment in the real numbers preceed as follows. Assume as given a function f_0 and for each i>0, make a choice of some ith function f_i depending on the previous choice of f_{i-1}. By carrying out this experiment with a transfinite mechanism, we can suppose that after ω iterations the sequence $\langle f_i : i \geq 0 \rangle$ of functions has been obtained. We would like to declare that experiment succeeds just in case the f_i converge uniformly to a function f. Future investigations should be able to decide whether or not this results in a meaningful statement about the semantics of PL.

REFERENCES

1. Bell, J. L. and Slomson, A. B., (1969), Models and Ultraproducts, North-Holland, Amsterdam.

2. Benson, D. B., (1970), "Syntax and Semantics: A Categorical View," Information and Control, 17, 145-160.

3. Cheatham, T. E. and Wegbreit, B., (1972), "On a Laboratory for the Study of Automatic Programming," Proceedings of an ACM Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, 208-211.

4. DeBakker, J. W., (1971), "Axiom Systems for Simple Assignment Statements," [9], 1-22.

5. DeMillo, R. A. and Chiaraviglio, L., (1971), "On the Applicative Nature of Assignment," Georgia Institute of Technology (School of Information and Computer Science) Research Report No. GITIS-71-01, Atlanta.

6. DeMillo, R. A. and Chiaraviglio, L., (1972), "The Semantics of Command Languages," (to appear in Philosophy of Exact Sciences: Problems, Methods and Goals).

7. DeMillo, R. A. and Chiaraviglio, L., (1972), "General Semantic Properties of First Order Programs," (to appear).

8. Elgot, C. C., (1971), "Algebraic Theories and Program Schemes," [9], 71-88.

9. Engeler, E. (ed.), (1971), Symposium on Semantics of Algorithmic Languages, Lecture Notes in Mathematics, Springer-Verlag, Heidelburg.

10. Engeler, E., (1967), "Algorithmic Properties of Structures," Mathematical System Theory, 1, 183-195.

11. Engeler, E., (1968), "Remarks on the Theory of Geometrical Constructions," Barwise, J. (ed.), The Syntax and Semantics of Infinitary Languages, Lecture Notes in Mathematics, Springer-Verlag, Heidelburg, 64-76.

12. Feldman, J. A., (1972), "Automatic Programming," Stanford AI Memo No. 160, Stanford.

13. Floyd, R. W., (1967), "Assigning Meanings to Programs," Schwartz, J. T. (ed.) Mathematical Aspects of Computer Science, Proceedings of a Symposium in Applied Mathematics, 19, American Math. Society, Providence, 19-32.

14. Hoare, C. A. R., (1969), "An Axiomatic Basis for Computer Programming," Communications of the ACM, 12, 576-583.

15. Ianov, I. I., (1958), "On Logical Schemes of Algorithms," Problems of Cybernetics, 1, 75-127.

16. Igarashi, S., (1964), "An Axiomatic Approach to the Equivalence Problems of Algorithms with Applications," Ph.D. Thesis, University of Tokyo, Tokyo.

17. Igarashi, S., (1971), "Semantics of ALGOL-like Statements," [9], 117-177.

18. Johnston, J. B., (1971), "The Contour Model of Block Structured Processes," Tou, J. T. and Wegner, P. (eds.), Proceedings of a Symposium on Data Structures in Programming Languages, University of Florida, Gainesville, 55-82.

19. Kaplan, D. M., (1968), "Some Completeness Results in the Mathematical Theory of Computation," Journal of the ACM, 15, 124-134.

20. Knuth, D., (1971), "Examples of Formal Semantics," [9], 212-235.

21. Landin, P., (1965), "A Correspondence Between ALGOL 60 and Church's Lambda Notation," Communications of the ACM, 8, 89-101 (Part I) and 8, 158-165 (Part II).

22. Lauer, P., (1971), "Consistent Formal Theories of the Semantics of Programming Languages," Technical Report No. TR25.121, IBM, Vienna.

23. London, R. L., (1970), "Proof of Algorithms: A New Kind of Certification (Certification of Algorithm 245 TREESORT 3)," Communications of the ACM, 13, 371-373.

24. Lucas, P. and Walk, K., (1969), "On the Formal Description of PL/I," Annual Review in Automatic Programming, 6, Pergamon, London.

25. Manna, Z., (1968), "Formalization of Properties of Programs," Stanford AI Memo No. 64, Stanford.

26. Manna, Z., (1969), "Properties of Programs and the First Order Predicate Calculus," Journal of the ACM, 16, 244-255.

27. Manna, Z., Ness, S. and Vuillemin, J., (1972), "Inductive Methods for Proving Properties of Programs," Proceedings of an ACM Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces.

28. Manna, Z. and Waldinger, R. J., (1971), "Toward Automatic Program Synthesis," [9], 270-310.

29. Manna, Z., (1968), "Termination of Algorithms," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh.

30. McCarthy, J., (1962), "Towards a Mathematical Science of Computation," Proceedings IFIP Congress 62, North-Holland, Amsterdam, 21-28.

31. McCarthy, J., (1963), "A Basis for a Mathematical Theory of Computation," Braffort, P. and Hirschberg, D. (eds.), Computer Programming and Formal Systems, North-Holland, Amsterdam, 33-69.

32. McCarthy, J., (1963), "Situations, Actions and Causal Laws," Reprinted in Minsky, M. L. (ed.), (1968), Semantic Information Processing, MIT Press, Cambridge, 410-417.

33. McCarthy, J., (1966), "A Formal Description of a Subset of ALGOL," Steele, T. B. (ed.), Formal Language Description Languages for Computer Programming, North-Holland, Amsterdam, 1-7.

34. McCarthy, J. and Painter, J., (1967), "Correctness of a Compiler for Arithmetical Expressions," Stanford AI Memo No. 40, Stanford.

35. McCarthy, J., (1968), "Some Philosophical Problems from the Standpoint of Artificial Intelligence," Stanford AI Memo No. 73, Stanford.

36. Mendelson, E., (1964), Introduction to Mathematical Logic, Van Nostrand, New York.

37. Morris, J. H. (1968), "Lambda Calculus Models of Programming Languages," Ph.D. Thesis, MIT, Cambridge.

38. Nauer, P., et. al., (1960), "Report on the Algorithmic Language ALGOL 60," Communications of the ACM, 3, 299-314.

39. Orgass, R., (1968), "Problems in the Theory of Programming," IBM Report No. RC 2236 #10411, Watson Research Center, Yorktown Heights.

40. Paterson, M. and Hewitt, C., (1970), "Comparative Schematology," Concurrent Systems and Parallel Computation: Project MAC Conference, ACM Conference Record, Woods Hole, 119-128.

41. Platek, R. A., (1966), "Foundations of Recursion Theory," Ph.D. Thesis, Stanford University, Stanford.

42. Poore, J. H., (1970), "Toward an Algebra of Computation," Ph.D. Thesis, Georgia Institute of Technology, Atlanta.

43. Rasiowa, H. and Sikorski, R., (1970), The Mathematics of Metamathematics, 3rd Edition, Monofrafie Matamatyczne, Warszawa.

44. Rescher, N., (1966), The Logic of Commands, Dover, London.

45. Roehrkasse, R. C., (1971), "Abstract Digital Computers and Automata," Ph.D. Thesis, Georgia Institute of Technology, Atlanta.

46. Scott, D., (1970), "Outline of a Mathematical Theory of Computation," Proceedings 4th Annual Princeton Conference on Information and System Sciences, Princeton.

47. Scott, D., (1971), "The Lattice of Flow Diagrams," [9], 311-366.

48. Strachey, C., (1966), "Towards a Formal Semantics," Steele, T. B. (ed.), Formal Language Description Languages for Computer Programming, North-Holland, Amsterdam, 198-220.

49. van Fraassen, B. C., (1971), Formal Semantics and Logic, MacMillan, New York.

50. Wegner, P., (1971), "Data Structure Models for Programming Languages," Tou, J. T. and Wegner, P. (eds.) Proceedings of a Symposium on Data Structures in Programming Languages, University of Florida, Gainesville, 1-54.

VITA

Richard Alan DeMillo was born in Hibbing, Minnesota on January 26, 1947. He graduated from Hibbing High School in 1965 and attended The College of Saint Thomas in Saint Paul, Minnesota from 1965 to 1969, earning the Bachelor of Arts degree in Mathematics from that institution. From 1969 to 1972 he attended the Georgia Institute of Technology, where he earned the Doctor of Philosophy degree in Information and Computer Science.

Mr. DeMillo was employed as a programmer in the Computing Center and as a teaching assistant in the Department of Quantitative Methods while a student at Saint Thomas. During the summers of 1969 and 1970 he was a research assistant at the University of California's Los Alamos Scientific Laboratory in Los Alamos, New Mexico. While at Georgia Tech Mr. DeMillo has been a research assistant and teaching assistant in the School of Information and Computer Science.

Mr. DeMillo has authored and co-authored papers and reports in natural language linguistics, formal languages, logic and programming languages. He is a member of the Association for Computing Machinery and the Association for Symbolic Logic.

Mr. DeMillo is married to the former Diane Hanson of Española, New Mexico. They are expecting their first child.